COMPUTATIONAL THINKING IN CHILDREN:

THE IMPACT OF EMBODIMENT ON DEBUGGING PRACTICES IN PROGRAMMING

by

Junghyun Ahn

Dissertation Committee:

Professor John B. Black, Sponsor
Professor Susan Lowes

Approved by the Committee on the Degree of Doctor of Education

Date 20 May 2020

Submitted in partial fulfillment of the
requirements for the Degree of Doctor of Education in
Teachers College, Columbia University

2020

ABSTRACT

COMPUTATIONAL THINKING IN CHILDREN: THE IMPACT OF EMBODIMENT ON DEBUGGING PRACTICES IN PROGRAMMING

Junghyun Ahn

Three studies were conducted to better inform how instructional design of educational programming for children impacts learning. In these studies, we focused on how unplugged debugging activities, which require correction of coding errors, affect skills related to computational thinking and personal attributes of children.

Study 1 observed debugging performance across varying degrees of embodiment (full and low) with a control group. To identify and rectify coding errors, children in the full embodiment group walked on a floor maze whereas low embodiment group manipulated a paper character using their fingers. Study 2 examined the effects of different degrees of embodiment when combined with either coding or narrative based language on computational thinking and self-efficacy. Children fixed coding errors on a

worksheet using coding language or narratives, then performed their revised code using full or low embodiment. Study 3 explored whether congruent or incongruent hand gestures incorporated with either direct or surrogate embodiment enhanced children's graphic and text programming, self-efficacy, and persistence. In the congruent gesture group, participants placed coding blocks in the same direction that the programming character moves whereas incongruent gesture placed coding blocks in a linear fashion. Direct embodiment is where the participant uses their finger to move a character whereas surrogate embodiment is where the researcher is controlled by the participant through verbal commands.

The results on computational thinking skills were: 1) Children performed better in debugging and problem solving using low embodiment; 2) Programming efficiency increased with the use of coding language; 3) Higher performance on graphic programming was found with incongruent gesture while transfer from graphic to text programming improved with surrogate embodiment. In personal attributes: 1) Significant interaction effect was found between hand gesture and embodiment on self-efficacy; 2) Higher persistence was exhibited from direct embodiment.

These findings between embodiment and development of computational thinking skills and personal attributes may be utilized in the unplugged learning environment. This is particularly relevant in supporting students to acquire basic computational thinking skills where relevant technology resources are not available.

© Copyright Junghyun Ahn 2020

All Rights Reserved

ACKNOWLEDGMENTS

I dedicate this dissertation first and foremost to my parents. They have always believed in me and have never stopped supporting my long journey. I would also like to thank to my brother's family for their love and support. Without them, I would have never arrived where I am today.

I would like to thank the following individuals who guided and supported my entire graduate life, providing valuable advice throughout the process: Dr. John Black, Dr. Susan Lowes, and Dr. Young-Sun Lee.

I would also like to thank all my friends, especially WH, Eunjoo Unni, Jung ssam, Song ssam, Maria, and Lee. Their friendship and encouraging words were essential to completing this project.

Lastly, I would like to thank The One for going through this process with me and for supporting me in every possible way. I could not have done this without your help and, for that, I am forever grateful.

JA

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

I – INTRODUCTION

Background

In the field of education, there is increasing enthusiasm for science, technology, engineering, and mathematics (STEM) education. Specifically, there is lively discussion in the Computer Science, Instructional Technology, and Learning Science communities regarding the topic of computational thinking (CT) (National Research Council, 2010).

Wing's (2006) influential article defined CT as a collection of mental tools that enables the individual to solve problems more effectively by thinking like a computer scientist. It received significant attention as an essential competency in the 21st century (Ananiadou & Claro, 2009; Binkley, Erstad, Herman, Raizen, Ripley, Miller-Ricci, & Rumble, 2012). This form of thinking involves the use of computer science concepts such as debugging, abstraction, remixing, and iteration to solve problems (Brennan & Resnick, 2012; Ioannidou, Bennett, Repenning, Koh, & Basawapatna, 2011; Wing, 2008). Computational thinking can also be considered fundamental for K-12 students because it requires ‘‘thinking at multiple abstractions’’ (Wing, 2006, p. 35). Most researchers agree that teaching computer programming can be a way to train CT (Grover & Pea, 2013; Kafai & Burke, 2013; Lye & Koh, 2014; Mannila, Dagiene, Demo, Grgurina, Mirolo, Rolandsson, & Settle, 2014) and to apply CT skills (Orr, 2009). Thus, many educators assert programming is a crucial topic to be integrated into K-12 education (Kafai &

Burke, 2013; Margolis, Goode, & Bernier, 2011; Resnick, Maloney, Monroy-Hernández, Rusk, Eastmond, Brennan, Millner, Rosenbaum, Silver, & Silverman, 2009).

Recent emphasis on integrating children's programming education into core curriculum solicits conceptual and pedagogical implications of STEM, programming, and computing education for younger students (Manches & Plowman, 2015). The goal of programming education is to give children a basic understanding of computer concepts, make them more discerning end users, and, potentially, innovative creators themselves (Scaffidi, Shaw, & Myers, 2005). As a response to provide cognitive practices through programming education, this paper presents an attempt to introduce young students to CT and enhance self-efficacy through programming education. Specifically, the studies focus on debugging in programming, which is a step-by-step problem solving procedure to identify problems and fix codes (Carver & Klahr, 1986).

Three debugging studies with kindergarten to 3rd grade participants are introduced in this paper. The purpose of the studies was to teach children computational skills and cognitive skills related to programming through embodied debugging activities to detect and correct programming errors. In addition to cognitive practice, by learning how to deal with errors during programming, it is expected that students gain confidence in their problem solving and programming abilities. Ultimately, such feelings of achievement can help overcome barriers and enhance self-efficacy, persistence, and resilience to failure. The series of studies are closely related as subsequent studies were designed by revising earlier studies based on research findings.

Study 1 examined how embodied instruction during unplugged debugging activities influences children's computational thinking, cognitive skills in the domain of

programming, and self-efficacy. Study 2 added different programming languages on top of embodied instructions incorporated in Study 1. Debugging activities in Study 3 were designed around various gestures and physical embodiment.

Computational Thinking for Young Students

Computational thinking is included in the Framework for K-12 Science Education (National Research Council, 2010) as a scientific practice that students should participate in learning. Dimensions of CT involve concepts (e.g., loops, conditions, subroutines), practices (e.g., abstraction and debugging), and computational perspectives (Brennan & Resnick, 2012) originated from computer science. In particular, computational concepts and practices can be applied to other disciplines such as science, mathematics, social science, biology, language arts, and engineering (Kafai & Burke, 2013; Lye & Koh, 2014). Scholars argue that CT needs to be taught outside of computer science beginning in kindergarten (Barr & Stephenson, 2011; Yadav, Zhou, Mayfield, Hambrusch, & Korb, 2011). For children, this can be a powerful cognitive skill that positively impacts other areas of their intellectual growth (Horn, Crouser, & Bers, 2012). Therefore, CT should be added to every child's analytical ability (Wing, 2006).

Yet, identifying effective instructional practices to promote understanding and engagement in CT with young learners is relatively new. The studies that do exist often examine tertiary students undertaking computer science courses (Katai & Toth, 2010; Moreno, 2012). In order to successfully introduce CT to young students, educators and researchers need to put effort into developmentally appropriate programming education.

Computational Thinking Practices through Debugging

Without guidance on the cognitive aspects of computational practices and computational perspectives (Grover & Pea, 2013), the programming experience may be non-educative as students are not actively reflecting on their experience (Lye & Koh, 2014). They could be merely in trial-and-error mode rather than thinking as they are doing (Biesta & William, 2003). In related literature, programming education tends to only focus on the creation phase of programming (Kelleher, Pausch, & Kiesler, 2007; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) rather than on providing cognitive practices (Tew, Dorn, Leahy Jr., & Guzdial, 2008). Therefore, planning for programming in K-12 contexts should consider students to be engage in the thinking-doing process rather than just doing. Therefore, to foster young students' CT in K-12 settings, we planned the studies around the debugging stage of programming that potentially support CT through thinking-doing through reflection and scaffolding.

Researchers propose reflection as an example to foster CT. To engage in reflection, students need to review and think about their programming process (Lye & Koh, 2014). Reflection also encourages the review of one's own learning performance (Søndergaard & Mulder, 2012; Yang, 2010). Based on the definition of reflection, the debugging process works closely with reflection, as it involves metacognition, which refers to "the conscious planning, control, and evaluation of one's cognitive process by deploying skills and application of knowledge" (Sternberg & Sternberg, 2016).

Debugging involves breaking down the final program into mini programs, making the given task manageable (reduction in degrees of freedom). This instructional support

can be beneficial for novice programmers who usually have difficulty relating different commands together (Robins, Rountree, & Rountree, 2003) as they "identify programming actions at the level of individual programming statements" (Lehrer, Lee, & Jeong, 1999, p. 247). For example, testing and debugging possibly aid program comprehension in computational practice (McCauley, Fitzgerald, Lewandowski, Murphy, Simon, Thomas, & Zander, 2008) as it requires parsing programming into mini chunks to debug. This allows participants to focus mainly on the critical features of causal relationships between the commands, which is vital in programming comprehension.

Debugging in Programming

To optimize learning that engages core cognitive practices of CT through reflection and scaffolding, this paper focuses on the debugging procedure in programming. Debugging is a highly complex and dynamic process to search for bugs and, more importantly, to achieve an overall task goal (Carver & Klahr, 1986; Law, 1998). In education, debugging, also known as systematic error detection, is widely recognized as comprising of CT (Grover & Pea, 2013), which offers a valuable learning opportunity for cognitive skill development, including problem solving, metacognitive skills, logical reasoning, and persistence (Goulet & Slater, 2009; Holbert & Wilensky, 2011).

Dealing with problems and seeking solutions through debugging are known to impact students' persistence and confidence on working with difficult problems. Prior literature examined and reviewed the relationship between self-efficacy and computer literacy, suggesting that self-efficacy is a successful component of programming learning

(Ramalingam & Wiedenbeck, 1998). Debugging will develop highly positive self-efficacy and a greater degree of persistence among students as they practice dealing with difficulties.

In short, debugging is effective in learning CT and students' persistence for working with difficult problems. The design and implementation of debugging studies need to focus on attributes that typically lead to coping with failure or frustration when students encounter difficulties.

Embodied Cognition for Computational Thinking

The central focus of the studies is to support CT skills for young students new to programming. Thus, the debugging interventions were grounded on an unplugged activity through embodiment. Embodied cognition emphasizes bodily engagement and perception in the development of knowledge and understanding of abstract concepts (Barsalou, 2008; Johnson-Glenberg, Birchfield, Tolentino, & Koziupa, 2014; Lindgren & Johnson-Glenberg, 2013). An embodied approach possibly assists learners to understand abstract concepts involving debugging and programming activities by providing authentic experiences. Research in programming education demonstrates that to understand abstract programming concepts, it is more effective to physically enact the programming scripts than to imagine them in one's mind (Black, 2010; Fadjo, Lu, & Black, 2009). Studies conducted to enhance CT skills among younger age groups in combination with mathematical thinking skills, such as number line estimation, number senses, and two-dimensional geometric shapes (Sung, Ahn, & Black, 2017; Sung, Ahn, Kai, Choi, &

Black, 2016), found that the use of embodiment and verbal cues in planning stages positively impact coding performance as well as mathematical understanding. In STEM education research, incorporating learning activities that involve high levels of embodiment (body movement) led to greater retrieval and retention than low levels of embodiment (touch based, clicking the mouse) (Johnson-Glenberg et al., 2014).

By adopting physical embodiment, the debugging interventions took place in an unplugged environment. Learning to program without a computer is known as 'unplugged' or 'offline' programming (Bell, Alexander, Freeman, & Grimley, 2009; Wohl, Porter, & Clinch, 2015). Unplugged experiences are often recommended for novices and young students because they require possibly the least amount of cognitive demand and technical knowledge (Kotsopoulos, Floyd, Khan, Namukasa, Somanath, Weber, & Yiu, 2017).

The studies implemented unplugged embodiment during the debugging process in young students' programming curriculum to enhance CT and confidence. As such, they aimed to find programming activities that are fun and effective for children newly exposed to programming by lowering obstacles that programming possesses.

Purpose of the Study

The purpose of this research is to investigate how unplugged debugging approaches that emphasize embodiment, along with programming language or gestures, can be designed to develop children's computational thinking, cognitive skills related to programming, and self-efficacy to overcome students' sense of failure. The process by which children equip themselves with CT skills is an increasingly important issue in educational research. Programming education has been used as a part of promoting CT skills. Decades of research on programming exists (Kelleher & Pausch, 2005), but educators continue to struggle with effective and measurable ways to assess programming instruction (Guzdial, 2014, October 15). For example, little effort has been paid to discover the pattern of debugging, the inevitable process of getting programs to work. Debugging is a complex cognitive skill which requires substantial cognitive capacity and is not directly taught in most programming curriculum (Klahr & Carver, 1988). Although students are exposed to debugging almost always during programming, they do not receive explicit instruction to identify the discrepancy, pinpoint the bug in the program, and correct it. A handful of research has attempted to compare patterns between novices and experts, but not for children (Ahmadzadeh, Elliman, & Higgins, 2005).

An instructional approach to debugging skill is embodied cognition. Research on embodied cognition in programming education demonstrates that in understanding abstract programming concepts, it is more effective to physically enact the programming scripts than to imagine them in the mind (Black, 2010; Fadjo et al., 2009). When it comes to debugging practice embedded in embodiment principles, it will explicitly help children

overcome barriers when faced with errors during programming. As such, these three studies were structured for children to explore debugging in more concrete ways by incorporating principles from embodied cognition. A student bodily acts on given programming scripts, finds the errors, fixes them, and tests them out. Instead of testing by watching a virtual character move in a programming application on a touch-based tablet, the student physically engages in the programming process. This is embodiment in the development of knowledge and understanding of abstract concepts. Moreover, an embodied approach with unplugged activities without electronic devices makes young students comfortable when introduced to CT skills. Consequently, these unplugged learning environments are expected to support students to acquire basic CT skills when relevant technology resources are not available.

Regardless of the significant promise of debugging during programming (Anewalt, 2008; Overmars, 2004), the challenge is how to structure instruction in a more effective way to benefit computational thinking, cognitive skills related to programming, self-efficacy, and persistence for novice programmers. Thus, the approach in this paper aims to structure developmentally appropriate debugging strategies, where debugging interventions incorporated (1) unplugged instructional embodiment, (2) different types of programming language, and (3) congruent or incongruent gestures.

Studies Summary

The goal of this research is to suggest possible instructional implications for enhancing CT and self-efficacy in programming education. In this paper, three studies

were designed based on STEM education, learning science, and cognitive science. After observing young students struggling with a programming application in an after-school classroom, the researcher came up with the idea of designing debugging studies. To help young students become better computational thinkers, the studies focus on debugging skills, mechanisms heavily dealt with in the field of computer science, yet, ignored in childhood education. Considering the multiple challenges of debugging, we need to refine our teaching methods by making programming concepts as concrete as possible through perceptual experiences.

Participants in the three studies were students from kindergarten to 3rd grade who attended an after-school program and summer coding camp at a New York City public school. Embodiment was implemented during the debugging interventions. The difference among the three studies was that Study 2 added different types of programming language and Study 3 added different gestures along with embodiment (Table 1).

Table 1

Study Design Comparison

	Study 1			Study 2		Study 3	
Condition1	Degree of Embodiment			Degree of Embodiment		Embodiment Type	
Group	Full	Low	Control	Full	Low	Direct	Surrogate
Condition2	None			Programming Language Type		Gesture Type	
Group				Programming Language	Narratives	Congruent	Incongruent

In detail, Study 1 was conducted with 1st and 2nd grade students to compare the effects of embodied activities among three degrees of embodiment (Full, Low, No Embodiment) toward CT and self-efficacy. Two forms of physical embodiment operationally defined by the researcher— Full embodiment and Low embodiment — were used during the debugging interventions. In the Full embodiment group, learners physically moved their entire bodies (i.e., walking on the floor maze) to perform debugged coding scripts, while learners in the Low embodiment group manipulated a paper character using their fingers following debugged coding scripts. Thus, the main difference between Full and Low embodiment was the degree of embodiment − embodying with the full range of the body (i.e., walking) or a small part of the body (i.e., moving a paper character with one's fingers).

Based on the results from Study 1, Study 2 kept the Full and Low embodiment conditions, which showed effective results. In addition, different types of programming language (coding language vs. narratives) were applied to the debugging activities. The second study was conducted with 2nd and 3rd grade students. Immediately after students debugged and revised a given code with errors on a worksheet with different types of programming language, they tested it and received feedback by implementing Full or Low embodiment conditions.

These preliminary studies revealed low embodiment had better outcomes than full embodiment, so Study 3 incorporated a different type of embodiment, namely, Direct embodiment and Surrogate embodiment. In direct embodiment, students moved a small toy on the worksheet maze similar to the Low Embodiment groups in Studies 1 and 2. In

surrogate embodiment, instead of moving their bodies, students verbally commanded the researcher to move a toy on the worksheet maze.

Research Questions

As part of an initiative to develop a CT learning environment for young students, this paper explores how embodiment and programming language during an unplugged debugging intervention influence students' computational thinking, cognitive skills in the domain of programming and self-efficacy. The following three studies incorporated various forms of embodiment, programming language, and gestures. This research builds on the interplay of social and cognitive aspects to optimize the learning of 21st century skills under the recent move for children to code in (in)formal educational settings.

The overarching research question is, "How can instructions and learning environments be made more effective and affordable for young students' CT as well as personal attribute development within programming education?" Each study was distinctively examined based on the intervention that was implemented (Table 1).

How do "specific intervention" and "combination of two interventions" during debugging activities affect kindergarten to 3rd grade students' outcomes related to CT (i.e., debugging and programming skills, problem solving strategies) and self-efficacy? The intervention used in Study 1 was different degrees of embodiment (Full vs. Low embodiment). Study 2 applied different types of programming language format (coding language vs. narratives) along with different degrees of embodiment that were used in Study 1. In the last study, the interventions during debugging activities were different

types of embodiment (Direct vs. Surrogate embodiment) and gestures (Congruent vs. Incongruent gesture).

Overview of the Thesis

This chapter explains the purpose and rational for the study. Chapter II reviews literature grounded in this study. The relevant literature on debugging instruction in programming education includes the definition of debugging, the impact of debugging in cognitive development (CT, problem solving and self-efficacy), research on embodied cognition theories including various gestures, and research in the area of different programming languages. Chapter III describes details of the three studies and their findings on debugging strategies in programming education. Lastly, a summary and discussion about how the findings are related to and differ from previous research are presented. Implications for education and the classroom and limitations of the studies are also discussed.

II – LITERATURE REVIEW

This chapter reviews the theoretical approach of the three studies, which include debugging in computer science, embodied cognition, and programming language. The literature guided the research design and measurement of the studies.

Debugging in Programming Education

Past experience with the LOGO programming language (Papert, 1980), which was unpopular in the classroom because of the lack of resources to incorporate it into K-12 core subjects (3), solicited a redesign of programming related curriculum for young learners. Debugging, a fundamental process of programming, is one example that needs further research. Debugging is a step-by-step problem solving procedure to identify problems and fix codes (Carver & Klahr, 1986), which comprise of CT. Debugging is among the most important actions leading to the development of logical thinking, problem solving, and social interaction skills (Sipitakiat & Nusen, 2012). When learners are highly engaged in programming to complete a task, they endeavor to correct errors which enable the program to work successfully. Such a process creates an ideal situation to develop a set of skills, such as problem articulation, teamwork, persistence, and other key abilities needed in most things in life (Sipitakiat & Nusen, 2012). A benefit of debugging is accelerating cognitive development in terms of logical reasoning, problem solving, metacognitive skills, and persistence. How to debug is rarely explicit during

programming, and it is not directly taught. The abstract nature and complexity of programming makes debugging hard for young students.

This section describes the theoretical background of debugging; reviews related research; addresses the advantage of debugging activities, such as the development of CT, problem solving, metacognition, and self-efficacy; and discusses how debugging practice can support children's cognitive development in programming education.

Review of Debugging Research on teaching debugging strategies examined effective instructional methods to assist learners' difficulties in facing errors while programming. Klahr and Carver's (1988) debugging model summarizes its process in the following steps: 1) Compare the goal location and the actual movement outcome to determine if debugging is necessary, 2) Identify the bug by describing the discrepancy between the goal and the actual outcome, 3) Pinpoint the bug in the program using various clues and techniques, and 4) Correct the bug and test the outcome. They proposed memory aids (e.g., posters with commands, debugging steps, and discrepancy-bug mapping) to help students learn and transfer debugging strategies (Klahr & Carver, 1988). Law (1998) conducted research to compare two debugging strategies, planned versus ad hoc debugging, when participants coped with program errors. Those debugging strategies were tested on either a self-generated program or other-written program where some logical errors were planted. Ad hoc actions were demonstrated to be the preferred debugging strategy, which was not significantly related to programming authorship (i.e., self-generated or other-written), and planning was found not that effective. Ahmadzadeh et al. (2005) analyzed patterns of debugging among novice programmer computer science students to find effective methods of teaching debugging suitable for all students. She

discovered that the major impediment is students do not acquire the skills to debug programs effectively, despite their understanding of programming. Others created technology integrated debugging tools (i.e., Robo-blocks, puzzles) that helped students learn debugging strategies by applying various design patterns abilities (Lee, 2015; Nusen & Sipitakiat, 2011).

Despite the recent growing trend in programming education for children, these recommendations are more applicable to adult learners and fail to assist young students. Teaching young students how to debug in an effective and measurable way needs more attention. Therefore, the goal of this research is to improve debugging teaching methods for children and, further, to positively nurture related cognitive skills of computational thinking, problem solving, and self-efficacy.

Debugging as Computational Thinking Practices

Computational thinking is integrating the power of human thinking with the capabilities of computers (Wing, 2006). Simply put, CT draws on concepts and practices that are fundamental to computing and computer science. The definition of CT encompasses a general analytic approach to problem solving, the design of systems, and an understanding of human behaviors (National Research Council, 2010; Wing, 2006, 2008). Thinking about data and ideas and using and combining these resources to solve problems are what enable people to think computationally.

The trend towards integrating CT and programming into the K-12 STEM curricula has emerged. Wing (2006) called for teaching CT as a required skill for 21st century success just as reading, writing, and mathematics are in the core curriculum. A

core practice of CT is tackling complex tasks by decomposing them into step-by-step sub goals (Wing, 2006). Both debugging and CT practice involve systematic attempts to identify problems and fixing codes using a step-by-step decomposed problem-solving procedure (Carver & Klahr, 1986). Debugging is listed as an example of a computational practice among three dimensions of CT: computational concepts, computational practices, and computational perspectives (Brennan & Resnick, 2012). It is suggested that "learning activities that allow students to discover and explain scientific relationships, predict events, and learn procedural skills enable them to better understand these subjects, predict behavior, and build CT skills" (Phillips, 2009). Given that CT is closely related to the debugging process, adopting meaningful debugging practices will result in training better computational thinkers.

Fostering computational practices and computational perspectives, as well as the transfer of these competencies for general problem-solving, need further research to integrate programming into K-12 curricula (Lye & Koh, 2014). As for the instructional implication to make computational perspectives more pertinent in K-12 settings, Lye and Koh (2014) proposed a constructionism-based problem-solving learning environment with information processing, scaffolding and reflection activities.

To successfully encourage students to be computational thinkers, foundational skills of CT must develop before learning a programming language (Lu & Fletcher, 2009). This implies that practicing CT skills through unplugged debugging activities before working on programming applications would be effective. To enhance the efficiency of debugging toward CT skills through three studies in this paper, young students participated in debugging activities at the begging of programming education

before they created their own projects with a programming application. Also, the debugging intervention was in the form of unplugged activities so students would focus mainly on debugging procedures and any potential disruption from digital devices would be excluded.

Metacognition of Debugging Process

Metacognition is cognition about cognition or knowing about knowing. It is achieved from the interplay among metacognitive knowledge, metacognitive experience, goals (or tasks), and actions (or strategies) (Flavell, 1979). Research on problem solving from metacognitive perspectives argues that when a person encounters a problem, it is important to step back and analyze what has been done (Wickelgren, 1974). Wickelgren (1974) observed that participants exhibited very little metacognition while solving problems, struggling, and largely failing to overcome barriers.

The debugging process is related to metacognition, which refers to "conscious planning, control, and evaluation of one's cognitive process by deploying skills and application of knowledge, such as thought engaging in learning processes (Sternberg & Sternberg, 2016) to make sense about a question, solve a problem, or achieve goals. The power of debugging is that "errors benefit us because they lead us to study what happened, to understand what went wrong, and, through understanding, to fix it" (Papert, 1980, p. 114). The metacognitive aspects of debugging are also connected with enhancing CT through thinking-doing, such as reflection and scaffolding (Biesta & William, 2003). As mentioned earlier, reflection requires students to review and think about their programming process (Lye & Koh, 2014). Reflection also encourages the

review of one's own learning performance (Søndergaard & Mulder, 2012; Yang, 2010). An example of scaffolding in programming education is breaking down the final program into mini programs, making the given task manageable (reduction in degrees of freedom). This instructional support can benefit novice programmers who usually have difficulty relating different commands together (Robins et al., 2003). As such, it is necessary to underscore the importance of developing metacognitive skills, which can possibly benefit debugging situations.

Self-Efficacy Development through Debugging

Self-efficacy, a specific form of self-confidence, is a person's prediction of how well he or she can perform a specific task (Bandura, 1977). These beliefs influence the extent to which individuals perceive their ability to master and feel competent when engaging in specific behaviors (Bandura, 1986). High self-efficacy is critical in problem solving because self-efficacy influences the use of cognitive strategies, the amount of effort put forth, the level of persistence, the coping strategies adopted in the face of obstacles, and the final performance outcome (Bandura, 1986).

Skill at debugging can increase a programmer's confidence. Self-efficacy is a successful component of programming learning (Ramalingam & Wiedenbeck, 1998). Student computer programming self-efficacy positively predicted performance (Ramalingam, LaBelle, & Wiedenbeck, 2004). Similarly, students' computer self-efficacy is related to their learning performance in computer-based learning environments (Moos & Azevedo, 2009). The debugging interventions in this paper were designed around unplugged learning environment with embodiment that is beneficial for self-

efficacy. According to Lambert and Guiffre (2009), unplugged experiences improved fourth grade students' confidence in mathematics and their perceived cognitive skills.

Therefore, these three debugging studies provide a framework to guide the design and implementation of programming education, focusing on the attributes that typically lead to positive socio-emotional development in children. The question that follows is how debugging instruction can foster a high level of self-efficacy and persistence beneficial to becoming a great problem solver (Lambert & Guiffre, 2009).

Remaining Questions about Debugging Instructions

Despite efforts to teach debugging skills to young students, how can debugging practice effectively prepare them to be a good computational thinker? Specifically, the complexity and abstract nature inherent in programming makes debugging even more challenging (Subrahmaniyan, Kissinger, Rector, Inman, Kaplan, Beckwith, & Burnett, 2007). Such abstract nature may contribute to the discrepancy between the excitement and the discouragement towards programming. In many STEM disciplines, traditional methods of teaching can lead to rote learning, little long-term retention, and significant gaps in conceptual understanding (Bransford, Brown, & Cocking, 2000). Alternative approaches include revision of classroom and teaching methods used in formal instructional settings (Bell et al., 2009) and the design of new learning environments in informal learning settings (Cooper, Dann, & Pausch, 2000). An example is the development of children's programming applications using a constructionist, project-based approach, which enables users to create stories, games, puzzles, and other projects with a block-based visual programming language.

Regardless of the significant promise of alternative approaches (Anewalt, 2008; Basawapatna, Koh, & Repenning, 2010; Overmars, 2004), the challenge is on how to structure instruction in a more effective way to benefit computational skills, related cognitive skills, self-efficacy, and persistence, and then to transfer such skills to later learning of a different programming language. As such, the three studies in this paper aimed to find programming activities that are fun and effective for young students by lowering obstacles of programming (Ko & Myers, 2004).

Embodied Cognition

The prevalence of digital creation tools for children such as graphical programming applications increases the opportunity to practice higher-level thinking. The plethora of mobile apps, software, and stand-along technologies intended for children tend to focus on basic academic skills, rather than content creation or high-level thinking (Flannery, Silverman, Kazakoff, Bers, Bontá, & Resnick, 2013). This paper explored studies on the effectiveness of embodiment for students in kindergarten to 3rd grade to learn and apply concepts of CT. Given the recent emphasis on CT in childhood education and the premise of embodiment in development and learning in childhood, the research in this paper implemented embodiment that can bridge CT with meaningful and playful activities.

Gestures are actions. People use action to think and that action changes thought (Casasanto & Dijkstra, 2010; Chu & Kita, 2008; Glenberg & Kaschak, 2002; Goldin-Meadow & Beilock, 2010; Hostetter & Alibali, 2008; Jamalian, Giardino, & Tversky,

2013; Schwartz & Black, 1996; Schwartz & Black, 1999). Gesture is connected with learning by reflecting an individual's knowledge state and altering people's cognitive state (Goldin-Meadow & Beilock, 2010). In particular, gesture has been shown to play an important role in problem solving. For example, when solving a spatial visualization task, spontaneous gesture helps thinking (Chu & Kita, 2011), as it influences the strategy chosen for problem solving (Alibali, Spencer, Knox, & Kita, 2011). Gesture also helps reasoning. For instance, adults produced gestures when making inferences and constructing a mental model to solve gear problems (Schwartz & Black, 1996).

To assist young learners to construct a better understanding of CT and nurture self-confidence through programming education, the following studies draw on theories from embodied cognition. Research on embodied cognition suggests that it is beneficial for children to interpret and construct complex concepts using physical experience.

This section reviews theories and research related to embodied cognition, such as instructional embodiment and perceptual congruency of gesture, and the implications of embodied cognition to support children' development of CT skills, problem solving, and self-efficacy through debugging.

Review of Embodiment in STEM Education

Embodied cognition emphasizes the use of bodily engaged action and perception in the development of knowledge and understanding of abstract concepts (Abrahamson & Lindgren, 2014; Barsalou, 2008; Johnson-Glenberg et al., 2014; Lindgren & Johnson-Glenberg, 2013). Research suggests that cognition is not necessarily amodal symbol systems independent of the body, but perceptually grounded (Barsalou, 2008). Embodied

cognition proposes that mental representations are constructed from bodily interaction with the physical world (M. Wilson, 2002) in that cognition is typically grounded in multiple ways, including simulations, situated action, social interaction, and the environment (Black, 2007). The use of action and perception in the development of knowledge and understanding in a formal learning environment is gaining traction among educators over the past few years.

Research on embodied cognition suggests that an understanding of something involves being able to create a mental perceptual simulation of it when retrieving the information or reasoning about it (Black, 2010, p. 3). This perspective underscores the perceptual experience of conceptual learning such as physically acting on, embodying concepts to a surrogate, or using gestures. Embodied cognition possibly reduces cognitive load, supports better memory retention, and enhances conceptual learning and problem solving skills (Black, Segal, Vitale, & Fadjo, 2012; Glenberg, 2010; Glenberg, Gutierrez, Levin, Japuntich, & Kaschak, 2004; M. Wilson, 2002). With the advancement of touch-based tablets (i.e., gesture-based computing) in education, there is much attention on the embodied process behind learning activities.

How does the instructional design of gesture-based technology integration benefit learners' knowledge construction during programming? An embodied approach can assist learners to understand abstract concepts involving programming activities by providing authentic experiences. Studies in programming education demonstrate that to understand abstract programming concepts, it is more effective to physically enact the programming scripts than to imagine them in the mind (Fadjo, Hong, Chang, Geist, & Black, 2010;

Fadjo et al., 2009). Studies by Sung et al. (2016), which applied physical activities to help children embody computational perspectives, substantiate these claims.

Previous research has found that producing gesture assists children to learn. For example, encouraging children prior to (Broaders, Cook, Mitchell, & Goldin-Meadow, 2007) or during a lesson leads to improved performance after the lesson (Goldin-Meadow, Cook, & Mitchell, 2009). Gesture effects on children's learning can: (1) alter people's cognition in a more direct way (Segal, 2011), (2) add another layer of meaning by presenting information in two modalities (i.e., visual and motor) (Jamalian et al., 2013), and (3) offload working memory and lighten cognitive load (Zhao, 2018).

A conceptual framework of embodied cognition informs new perspectives of applying technology into learning. Creating learning activities utilizing technologies does not merely mean engaging in activities through a bodily state. Rather, students should meaningfully interact with perception, action, and the environment to accomplish the benefit of the activities. Recent research in embodied cognition claims the design of instructional embodiment provides a more meaningful learning environment as it makes learners engage in movement, imagination, and exploration.

Instructional Embodiment

To address pedagogical approaches to teach abstract principles of embodiment within formal instructional settings, Fadjo and his colleagues (Black et al., 2012; Fadjo et al., 2010) proposed "instructional embodiment." This conceptual framework consists of two primary categories that integrates both physical embodiment (i.e., direct, surrogate, augmented, and gesture embodiment) and imagined embodiment (i.e., explicit

embodiment, implicit embodiment). Within the four forms of physical instructional embodiment, the three studies were based on direct, surrogate, and gesture embodiment. Grounded on direct embodiment in which people move their body directly, the two preliminary studies implemented different degrees of embodiment – Full embodiment and Low embodiment. The third study employed gesture embodiment and type of embodiment – Direct embodiment and Surrogate embodiment ─ during the unplugged debugging intervention.

Degree of Embodiment: Full vs. Low Embodiment. Studies 1 and 2 derived concepts from two forms of direct embodiment ─ Full embodiment and Low embodiment ─ during the debugging intervention to examine their effect on CT and self-efficacy development in young students. The researcher operationally defined full embodiment as learners physically moving their entire bodies (i.e., walking on the floor maze) to perform coding scripts in which they revised errors. Another degree of embodiment was low embodiment, where learners manipulated a paper character using their fingers followed by debugged coding scripts. Thus, the main difference between full and low embodiment was the amount of embodiment, for example, embodying with the full range of the body (i.e., walking) or a small part of the body (i.e., moving a paper character with one's fingers).

Type of Embodiment: Direct vs. Surrogate Embodiment. According to Fadjo's conceptual framework (2012), direct embodiment refers to when the learner physically enacts a scenario of statements or sequences using his or her own body that

include explicit or implicit cues for movement (Fadjo, Lu, & Black, 2009). In Study 3, participants in the Direct embodiment group tested revised codes of pre-defined programming scripts on a worksheet by physically embodying the scripts. Surrogate embodiment is a type of physical enactment controlled by the learner where the manipulation of an external 'surrogate,' an agent designed to represent a learner (Fadjo, Shin, Lu, Chan, & Black, 2008). In Study 3, participants in the Surrogate embodiment group tested out revised code through a surrogate. They gave a verbal command to a human surrogate (the researcher) to move around on the floor. Compared to low embodiment in the two preliminary studies, where students manipulated a paper character on a worksheet with their finger, students did not physically move the surrogate, but gave commands to a human surrogate.

Gesture Embodiment: Perceptual Congruency vs. Incongruency. Gesture embodiment is characterized as the trained movement of the hands in connection with a particular physical enactment during an explicit learning activity. Gestures promote learning when they are compatible with learning content they represent. This form of action is called congruent physical action. Segal (2011, p. 1) argued that the "right" gestures should be congruent with learning concepts and compatible with the mental representation and operations needed to solve problems because thinking can be internalized action and that re-externalizing the thinking as congruent action can facilitate the thinking (Segal, 2011; Segal, Tversky, & Black, 2014). Similarly, in spatial and diagrammatic representation, congruent action has been known to support and affect

thinking (Goldstone, Landy, & Brunel, 2011; Hegarty, 2011; Kirsh, 1995; Tversky, 2011).

While congruent action demonstrates effective learning in many cases, the effect of incongruent action in learning shows mixed results. For example, Goldin-Meadow and Beilock (2010) found that performance was hindered when incompatible action was used. They argued that when perception and action contradict, it interferes with thinking and performance.

In contrast, some argue that excessive structure may interfere with critical cognitive processes that support the encoding of robust memory representations of the target concept (R. A. Bjork, 1994). Rather, some degree of failure, or "desirable difficulties," is necessary in the learning environment (E. L. Bjork & Bjork, 2011; R. A. Bjork & Linn, 2006). Several studies which explicitly incorporated inconsistency or ambiguity into the learning materials revealed greater higher-level thinking than simper materials (Byrge & Goldstone, 2011; Mannes & Kintsch, 1987; Martin & Schwartz, 2005), arguing that seemingly inconsistent or ambiguous materials promote transfer. That is, unchallenging instruction tasks produce successful performance during learning, but may limit long-term retention and transfer (Bjork, 1994; Bjork & Bjork, 2011; Bjork & Linn, 2006). A possible explanation for successful transfer is that the challenge of coordinating disparate materials (e.g., graphical model and symbolic representation) prompts deeper cognitive processes (Vitale, 2012) or "knowledge integration" (Clark & Linn, 2003). On the other hand, materials in which structural organization of the target concept was conveyed through intuitive design leads to decreased transfer. Explicit

design possibly fosters overconfidence and discourages learners from engaging in deeper reflection of the material (Vitale, 2012).

Gestures during Children's Programming. During the programming task, children created programming and problem solved to create interactive animations and stories. The design of most programming software developed for young learners asks to drag and drop coding blocks into a scripting area to activate them. Snapping coding blocks together in the scripting area creates programs that are read and played. They are created and run from left to right or top to bottom, the same way as written English. Yet, left to right or top to bottom features seemingly "incompatible" representations with the direction of the sprites (a.k.a., an independent computer graphic object which may be moved on-screen by programming command).

We examined how congruent gesture and incongruent gesture affect CT skills, its transfer, and self-efficacy in Study 3. In the study, congruent gesture is defined as when the coding block arrangement on the worksheet (i.e., action) is compatible with the movement of the sprite on the maze (i.e., perception). On the other hand, incongruent gesture is when the coding block arrangement on the worksheet contradicts the movement of the sprite on the maze.

During unplugged debugging activities, children were asked to revise the direction of the maze task. They either received a congruent or incongruent worksheet depending on the experimental group they were assigned. For the congruent group, the coding scripting area on the worksheet physically matched the direction of the maze, whereas the coding scripting area on the worksheet for the incongruent group did not

match the movement on the maze. The incongruent group has the same on-screen coding scripting area of most children's programming software (e.g., drag and drop coding black from left to right), which is incompatible with users' perception about the sprite's movement.

Debugging Instructions from Embodied Cognition

This section reviews the theoretical and empirical foundation of embodied cognition and summarizes the implications of embodied cognition in STEM education for young students. The literature reviews how the use of body influences perceptual understanding and learning and addresses how bodily engagement develop learners' perceptual experiences in STEM disciplines. Opportunities for embodied cognition to enhance learning outcomes in the advancement of gesture-based computing (i.e., touch-based tablet) are also addressed. Drawing on literature in this section, embodied cognition can be a powerful framework for the creation of a learning environment in programming education for young learners.

The following studies were built upon embodied cognition for the purpose of creating potentially valuable programming learning environments to enhance children's CT skills, cognitive skills in the domain of programming, and positive self-efficacy through an embodied approach with debugging activities. Evidence from this review indicates that instructional design that integrates kinesthetic practice can lead to meaningful perceptual experiences, which, ultimately, results in higher conceptual learning outcomes as well as confidence and persistence in children.

Programming Language Forms

Along with the embodied debugging intervention in an unplugged environment in the three studies, Study 2 examined different types of programming language that are similar to everyday language. Traditionally, most programming is based on programming syntax, which makes it difficult for children to understand (Wang, Qi, Zhang, & Wang, 2013). To reduce the challenges of working with programming syntax among children, visual programming (e.g., moving coding blocks on the touch-based tablet screen) was introduced. Compared to traditional programming languages (i.e., Java or C++) that resembles the computer's way of thinking (Smith, Cypher, & Tesler, 2000), visual programming languages mostly found in children's programming applications use representation that is closer to human language (Lye & Koh, 2014). According to Lye and Koh (2014), features in visual programming, such as users dragging and snapping the command blocks, help reduce the cognitive load on students and help them "focus on the logic and structures involved in programming rather than worrying about the mechanics of writing programs'' (Kelleher & Pausch, 2005, p. 131). Therefore, visual programming language can potentially allow students to acquire computational concepts more easily.

However, dealing with symbols on coding blocks and manipulating them simultaneously in visual programming can be overwhelming and often discouraging for novice programmers (Kelleher & Pausch, 2005). Writing program syntax with an on-screen application can be more difficult than visual programming that involves direct manipulation or form filling, but often gives the student more power of thinking. Kelleher and Pausch (2005) examined natural everyday language as a different type of written

format of programming language. In their study, reducing mechanical difficulties by programming with everyday language instead of programming syntax improved novice programmers' first programming experience. Another study confirmed that programming languages that match users' natural vocabulary and expressions of computation significantly increase their ability to complete programming problems (Pane & Myers, 2006). These studies confirm that programming language should be more similar to users' spoken language rather than systemic computer syntax. Yet, how programming language that resembles natural spoken language impacts programming learning has not been investigated with young learners.

Therefore, Study 2 examined two different types of languages that are closer to spoken English. It aimed to reveal the relationship between programming and CT skills, cognitive skills in the domain of programming, as well as self-efficacy of young students through embodiment and programming languages by placing emphasis on the debugging stages that young students are not familiar with.

III – STUDY ONE

Overview

The challenges associated with the growing attention on children's programming with gesture-based computing require the development of quality programs that provide a learning environment that cultivates computational thinking (CT) skills and confidence towards self. The goal of this study is to teach children the concepts of programming and debugging by introducing how to detect errors and correct to reach a given goal. By practicing how to deal with errors with paper-based debugging tasks, it is expected that they will become more knowledgeable about programming and self-confident.

Research Questions and Hypothesis

In the first study, children participated in an unplugged debugging intervention designed according to theories and studies on embodied cognition. Children's activities with the Scratch Jr. programming software during the debugging sessions were analyzed to determine the effect of CT, problem solving skills related to programming, and influence on the children's confidence on self. Specifically, this study investigated the following research questions:

1) What is the effect of different degrees of embodiment (Full embodiment, Low embodiment, Control group) implemented during a paper-based debugging

intervention on debugging abilities in relation to computational thinking and self-efficacy?

In comparing the different embodiment groups (Full, Low, and Control group), it is hypothesized that children receiving any kind of embodiment will have better acquisition of CT and problem solving skills than those in the control group. Within the embodiment condition, it is hypothesized that low embodiment will strongly support CT and problem solving skills. For self-efficacy, it is hypothesized that full embodiment will be strongly related to self-efficacy, as these children may feel some kind of authorship on the debugging task by physically moving their body (Ahmadzadeh et al., 2005).

Participants

Participants were 51 kindergarten and 1st grade students from three classes of an after-school program at a New York City public school. Children attended only two sessions of a Scratch Jr. programming class before the debugging study began, so they had a very basic understanding of how the Scratch Jr. software coding blocks work. No other programming language was formally introduced to the students before they began the program. Before implementing the study, the participants were already assigned to after-school coding classes and summer camp coding classes by the school's administration. Thus, the study recruited participants following a quasi-experimental design (Shavelson & Towne, 2002) instead of random assignment. In each class, the same researchers taught.

Research Design

The three after-school classes were assigned to three different type of embodied intervention groups ─ Full embodiment, Low embodiment, and a Control condition (Table 2).

Table 2

Experimental Groups by Degrees of Embodied during Unplugged Debugging Intervention

Degree of Embodiment		
Full Embodiment	Low Embodiment	Control Group
$N = 19$	$N = 18$	$N = 14$

Specifically, children in the Full embodiment group used their own bodies to physically enact a script of given coding blocks in the maze created on the floor (i.e., walking on the floor maze) as seen in Figure 1. In the Low embodiment group, children manipulated an external surrogate (e.g., paper octopus) to simulate a script of given coding blocks on a worksheet (Figure 2). Those in the Control group worked on a paper without any external surrogate, limited to the use of a pencil (Figure 3).

Figure 1. Full embodiment group using floor maze

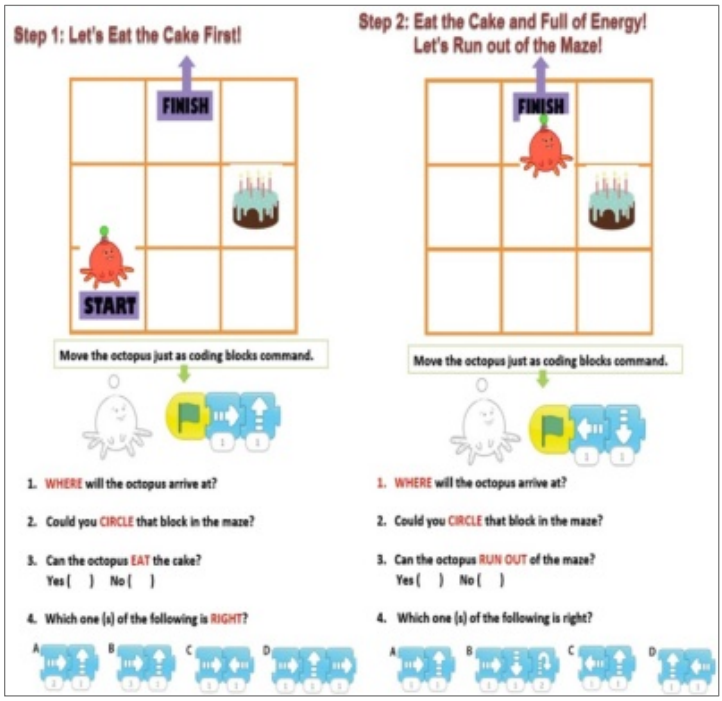


Figure 2. Low embodiment group using a paper octopus on a worksheet

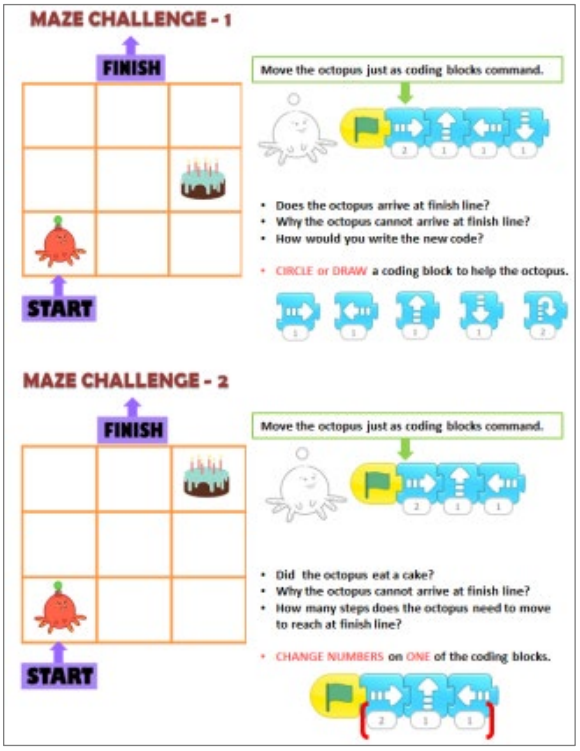


Figure 3. Control group working on a worksheet

Procedures

This study consisted of an embodied debugging activity on a floor maze under three intervention conditions. Then, an iPad debugging assessment session using Scratch Jr. children's programming software followed. While participating in the debugging intervention, which asked children to help a character get out of the maze created on the classroom floor, each individual student was presented with a multiple choice of coding blocks. They were then asked to judge if a given coding block would fix bugs the character encountered in the maze to reach the end point according to the intervention group they assigned. A researcher provided scaffolding questions, if needed, to guide the

children to fix the bugs successfully by comparing the goal location and the current location the character was standing.

Prior to the start of the iPad debugging intervention, four intentional errors (bugs) were planted in the iPad debugging post-test by the researcher (Figure 4). Following the successful completion of the embodied debugging intervention for each child, each debugged a pre-programmed debugging task in Scratch Jr. as a post-test. The iPad debugging task was designed to test overall debugging skills while each bug was designed to test specific debugging strategies necessary to program. These bugs were based on four error-prone computational concepts and skills (Brennan & Resnick, 2012; A. Wilson & Moffat, 2010) including: 1) incorrect type of movement regarding the object, 2) incorrect number of steps regarding object definition, 3) incorrect message sent or received regarding thread synchronization, and 4) missing initiating connection between two characters regarding collision detection. A self-efficacy test was administered before the embodiment intervention and immediately after the iPad debugging session.

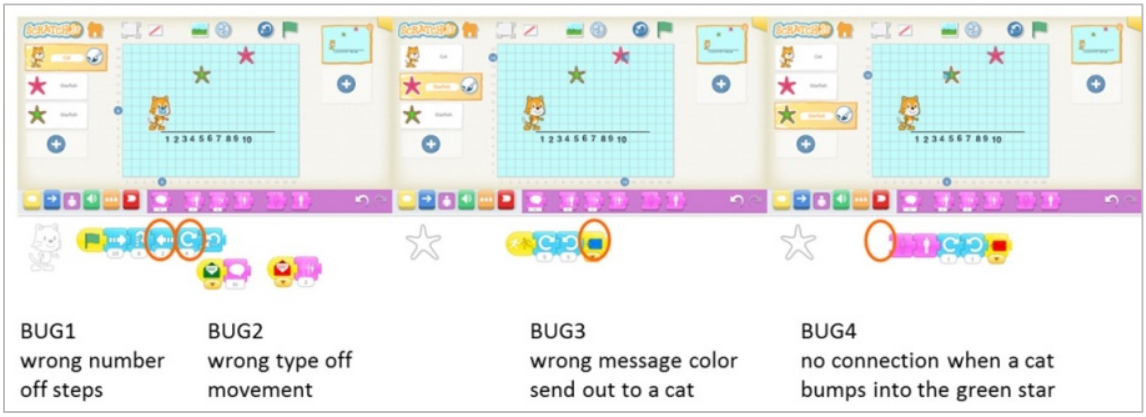


Figure 4. Four errors in the pre-programmed Scratch Jr. iPad debugging post-test

The iPad debugging session was videotaped and recorded each student's iPad screen and audio of their think-aloud during the debugging post-test. During the analysis of the screen recordings, we monitored students' process of debugging to examine strategies used to achieve the overall task goals rather than analyze their final artifact. Overall debugging performance was calculated by the number of bugs identified among the four types of pre-programmed bugs (on a scale from 0 to 4). Overall programming proficiency was scored on a scale from 0 to 2, as the sum of coding efficiency (ability to come up with a short route scored as 1, inability scored as 0) and coding fluency (instead of using the same coding block several times consecutively, the number of steps used in one coding block to represent same the block, scored as 0 or 1) (Klahr & Carver, 1988; Wyeth, 2008). Students' rewriting strategy, where they partially or entirely deleted a given code and wrote new code to reach goals, was scored as 0 or 1 (use of rewrite strategy scored as 1, no use scored as 0). Children's perception of self-efficacy was measured by six survey items made up of questions from the Positive Technology Development Questionnaire (Bers, Doyle-Lynch, & Chau, 2012) and AIR Self-Determination Scale (Wolman, Campeau, Dubois, Mithaug, & Stolarski, 1994), based on a 5-point Likert scale (Appendix A).

Results

When comparing the groups on debugging performance, students in the Low embodiment group had the highest debugging performance score while those who in the Control group scored lowest. On the programming proficiency post-test, the Low

embodiment group had the highest score and the Control group scored the lowest (Figure 5). One-way ANOVA tests (Table 3) were conducted to compare the effect of the embodied debugging intervention on overall debugging performance and programming proficiency. Results showed significant differences were found in debugging performance, F (2, 48) = 6.167, p = .004, η^2 = .204, and also in programming proficiency, F (2, 48) = 4.762, p = .013, η^2 = .166. Further post-hoc analysis indicated that the Full embodiment group (p = .013 for debugging performance) and the Low embodiment group (p = .006 for debugging performance, p = .009 for programming proficiency) significantly outperformed the Control group (Table 4).

Table 3

One-Way ANOVA of Debugging Performance and Programming Proficiency by Group

Source		SS	df	MS	F	Sig.	Partial Eta Squared
Debugging Performance	Between Groups	15.232	2	7.616	6.167	0.004**	0.204
	Within Groups	59.278	48	1.235			
	Total	74.510	50				
Programming Proficiency	Between Groups	4.973	2	2.487	4.762	0.013*	0.166
	Within Groups	25.066	48	0.522			
	Total	30.039	50				

* $p < .05$, ** $p < .01$

Table 4

Post-hoc Test for Debugging Performance and Programming Proficiency by Group

DV			MD	SE	Sig.	95% CI	
						Lower Bound	Upper Bound
Debugging Performance	Full	Control	1.15	0.391	0.013*	0.21	2.10
		Low	-0.13	0.366	0.931	-1.02	0.75
	Low	Control	1.29	0.396	0.006**	0.33	2.24
		Full	0.13	0.366	0.931	-0.75	1.02
Programming Proficiency	Full	Control	0.41	0.255	0.245	-0.20	1.03
		Low	-0.38	0.238	0.256	-0.95	0.19
	Low	Control	0.79	0.258	0.009**	0.17	1.42
		Full	0.38	0.238	0.256	-0.19	0.95

* $p < .05$, ** $p < .01$

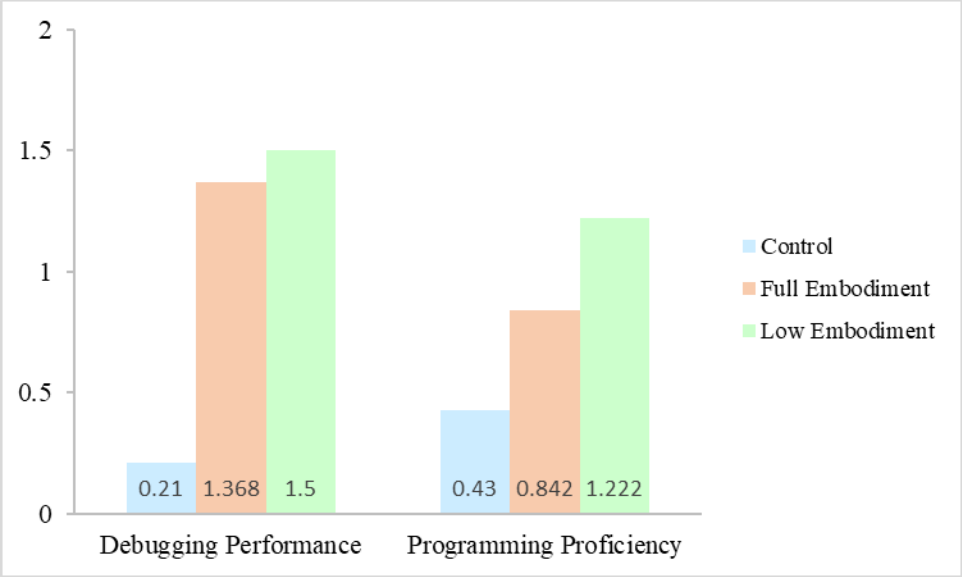


Figure 5. Mean comparison on debugging performance and programming proficiency

The "delete and rewrite all" strategy, where students partially or entirely delete pre-program debugging and rewrite the code to fix the errors during Scratch Jr. post-test, was adopted by 65% of the participants even though two other strategies (replace some of

the given codes and add more to compensate) were presented in the embodied debugging intervention. Results from a logistic regression indicate that the Low embodiment group rewrote the given code significantly more than the Full embodiment group ($p = .036$). While the Low embodiment group tended to rewrite more than the Control group, no significant differences were found (Table 5).

Table 5

Logistic Regression Analysis of Delete or Rewrite Strategies during Debugging Post-test

Embodiment Group	β	SE β	Wald's χ^2	df	Sig.	e^{β}
Low vs. Full	1.646	0.78	4.398	1	0.036*	0.193
Low vs. Control	0.730	0.87	0.695	1	0.405	2.074
Full vs. Control	-0.92	0.76	1.467	1	0.226	0.400
n =	49					

* $p < .05$

Self-efficacy pre-test scores showed no significant difference among the groups. The magnitude of gain scores (score difference between post-test and pre-test) was calculated and a one-way between subjects ANOVA was performed to compare the effect of the embodied pre-programming activity on self-efficacy in the Full, Low, and Control groups (Table 6). Only one item, "Having confidence using a computer," was significant among the three groups, $F(2, 33) = 3.485$, $p = .042$, $\eta^2 = .174$. Further post-hoc analysis showed that the Full embodiment ($p = .028$) and Low embodiment groups ($p = .035$) were significantly more confident using a computer than the Control group, while there was no significant difference between the Low and the Full embodiment groups or between the Full and Control groups.

Table 6

One-Way ANOVA of Self-Efficacy on Computer Usage by Embodiment Group

Source	df	SS	MS	F	Sig.	Partial Eta Squared
Between Groups	2	5.401	2.700	3.485	0.042*	0.174
Within Groups	33	25.571	0.775			
Total	38	30.972				

* $p < .05$

Discussion

This study examined the benefits of embodied instruction for children to develop debugging and programming skills which comprise of computational thinking skills and self-efficacy in the domain of programming education. Findings suggest that both Full and Low embodiment groups outperformed the Control group on debugging skills and tended to have higher self-efficacy in computer usage. Those in the Low embodiment group scored higher than those in the Control group on programming proficiency.

Although there are efforts to make programming more appealing (Bers, Flannery, Kazakoff, & Sullivan, 2014) and more accessible to learners (Kelleher & Pausch, 2005), few studies specifically examine effective, measurable ways to teach programming (Guzdial, 2014) that are more accessible and error-tolerant (Ko, 2009) to children. This leads to a new approach in pedagogical design, where young learners utilize embodied instruction to practice debugging strategies to keep them interested rather than discouraged when facing errors while programming.

One of the limitations in this study is the small sample size, decreasing the power of the study. Also, students in the Control group were allowed to use a pencil when they worked on the debugging worksheet. This can be considered a type of surrogate, which can be major flaw of the study. In addition, each phase of the study took place only once a week for an hour each week. This may have weakened the effect of the results of the study. Despite these limitations, the findings provide effective and measurable ways to teach debugging curriculum that nurtures CT, problem solving, and metacognition. Furthermore, cultivating positive self-efficacy in programming skills can be a crucial asset in the acquisition of programming skills.

IV – STUDY TWO

Overview

This study examined how different degrees of embodiment (Full embodiment where learners move their body to enact code versus Low embodiment where learners manipulate a paper character) and text-based programming language (Coding language versus Narratives) during a debugging activity affect elementary students' CT skills, cognitive skills related to CT, as well as self-efficacy.

Research Questions and Hypothesis

Specifically, this study examined the following research questions:

Research Question 1: How do different degrees of embodiment during debugging activities affect 2nd and 3rd grade students' outcomes of debugging and programming skills, cognitive skills related to CT, and self-efficacy?

Hypothesis 1: It is hypothesized that low embodiment better promotes participants' debugging and programming skills compared to full embodiment, while full embodiment, in which participants physically move their bodies, will lead to higher self-efficacy because of the feeling of authorship on debugging (Ahmadzadeh et al., 2005).

Research Question 2: How do different type programming language format during debugging activities affect 2nd and 3rd grade students' outcomes of debugging and programming skills, cognitive skills cognitive skills related to CT, and self-efficacy?

Hypothesis 2: When a text-based programming language is applied to the debugging intervention, coding language will be positively related to debugging and programming skills, and cognitive skills as students specify a more detailed sequence of routes by which the character on the paper-based debugging task will proceed (Hayes-Roth & Hayes-Roth, 1979; Webb, Ender, & Lewis, 1986). In contrast, participants using narratives to revise the coding script will have a more favorable attitude towards self-efficacy as it is closer to daily spoken English, thus, they will exhibit more comfort while working on the debugging task.

Research Question 3: What combination of degree of embodiment and type of programming language yields better learning to debugging and programming skills, cognitive skills cognitive skills related to CT, and self-efficacy?

Hypothesis 3: Given the combination of embodiment and text-based programming factors, it is hypothesized that those in the Low embodiment with Coding language group will have the greatest benefit for debugging and programming tasks and cognitive skills related to CT. Those in the Full embodiment with Narratives group will have higher self-efficacy.

Participants

Participants included 59 students from two after-school program classes and two summer camp programming classes, with 2nd and 3rdgraders mixed in each class. Students were recruited from a daily after-school program and a week-long summer camp at a New York City public school that serves a majority of low-income students. Students

consisted of Hispanic, African American, and English as Second Language learners. Over half of the participants were male (57.6%).

Research Design

This study was conducted in three sessions for 50 minutes per week in a NY public elementary school as part of an after-school curriculum. Participants engaged in unplugged debugging activities using a maze problem that had errors they needed to find and revise in order to successfully get out of the maze. Participants were randomly assigned to one of four conditions with two factors (Table 7) ─ type of programming language (Coding language or Narratives) and type of embodiment (Full or Low embodiment) ─ to fix and revise the code for directions in a given maze problem.

Table 7

2 x 2 Factorial Experimental Group

Type of Programming Language	Degree of Embodiment	
	Full Embodiment	Low Embodiment
Coding Language	Group 1: Full Embodiment with Coding Language (N = 16)	Group 2: Low Embodiment with Coding Language (N = 13)
Narratives	Group 3: Full Embodiment with Narratives (N = 15)	Group 4: Low Embodiment with Narratives (N = 15)

The first intervention condition was type of programming language (Coding language vs. Narratives) (Figure 6). Each programming language group received different paper worksheets that they needed to find errors and revise. Both group's mission was the same - to find and fix errors for a character to get out of the maze on the worksheet - even though the language description differed. The Coding language group received a paper worksheet written in programming syntax-like code while those in the Narratives group received a descriptive one similar to their everyday language.

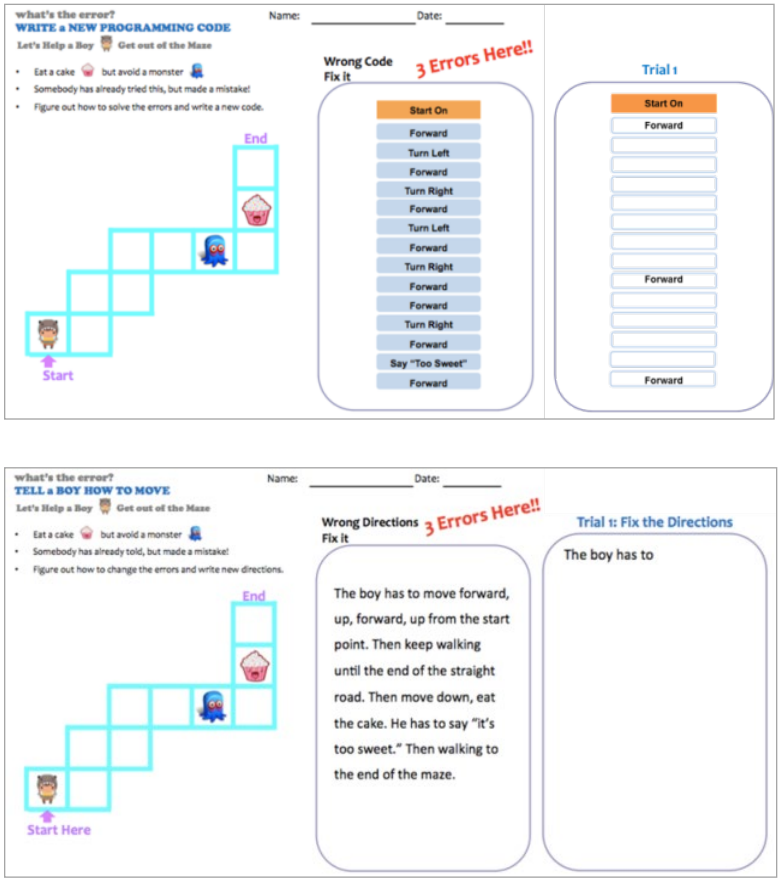


Figure 6. Coding language worksheet and narratives worksheet

The second intervention condition was degree of embodiment. (Figure 7). After fixing errors on the worksheet through coding language or narratives, learners in the Full embodiment condition physically moved their entire bodies (i.e., walking on the floor maze) to perform the revised coding scripts. In the Low embodiment condition, learners manipulated a paper character using their fingers and following the revised coding scripts. Thus, the main difference between Full and Low embodiment was the degree of embodiment, for example, embodying the full range of the body (i.e., walking) or a small part of the body (i.e., moving a paper character with one's fingers).

Figure 7. Full embodiment and low embodiment

During the debugging intervention, students in each class were randomly assigned to one of four intervention groups that differed in type of programming language (Coding language or Narratives) and embodiment (Full or Low embodiment) to fix and revise code or directions in the given maze problem (Figure 6 & 7).

Participants in the Full embodiment with Coding language group used coding language to revise the given coding errors and then physically enacted the revised code with their own bodies. In contrast, in the Full embodiment with Narratives group, participants used narrative text to revise the given maze then physically enacted the revised code. While both groups in the Full embodiment condition were prompted to physically move their bodies to enact the revised code, the groups in the Low embodiment condition simulated the revised code with a paper character that acted as a surrogate role for the participants. Those in the Low embodiment with Coding language group used coding language to revise the given errors and wrote new code. They then moved the paper character on the worksheet using their revised directions. However, participants in the Low embodiment with Narratives group used narrative text to revise the given maze directions, then moved the paper character on the worksheet using their revised directions. In each classroom, the same researchers taught and collected the data.

Materials

After students participated in the debugging intervention, four post-tests were administered. Post-tests consisted of a paper-based debugging post-test, an electronic programming post-test, the Test of Problem Solving (TOPs-3): Elementary interview, and a self-efficacy survey on coding activities, computer usage, and school behavior.

Programming Debugging Test

The paper-based post-test was developed by the researcher to determine whether participants improved their debugging skills after the intervention (Appendix B). The 10 questions were derived from essential programming strategies such as sequence, pattern recognition (repeat), and conditional. For six of the questions, participants filled in the blank for the problems which asked them to revise given codes (sample code that they need to use was provided) or write the number for a 'repeat' programming function. The remaining four questions were multiple choice items to determine the right code for the character's final destination after coding was executed. Participants had as much time as they needed to complete this test. The number of correct answers was used for data analysis.

Programming Test

This electronic test was administered on an iPad using a basic level game in the Tynker programming software. With Tynker, the user drags and drops a text-based coding block to create coding script to make the character move to accomplish the game mission. This software was selected because it is age appropriate for elementary school students and saves the user's information about what levels were played and how efficiently the programming strategies were applied. To assess whether the intervention was helpful to efficiently create programs, the level they reached and efficiency scores were used for data analysis. There are 20 levels in the basic Tynker game with up to three programming efficiency scores. Participants played this game for 20 minutes.

TOPs-3 Interview: Cognitive Skill Test

To measure cognitive skills related to CT, an interview with individual participants using the Elementary Test of Problem Solving (TOPs-3) (Bowers, Huisingh, & LoGiudice, 2005) was administered. Participants were shown photos of scenarios followed by a standard set of questions read aloud by the researchers. For this study, three pictures were selected from the TOPs-3 elementary package. Questions associated with the three pictures related to sequence, problem solving, predicting, or determining cause, which comprise cognitive skills. Responses were scored based on the scoring guide. Responses under each category were summed to measure each skill. The sum of the four subcategories was used as a general cognitive skill score.

Self-Efficacy Survey

Participants' self-efficacy, belief about self, in working with coding software and computers and behavior in the classroom were measured after the debugging intervention. This 5-point Likert scale survey consisted of 15 survey items (Appendix C). Five survey items were selected from the Positive Technology Development Questionnaire (Bers, Doyle-Lynch, & Chau, 2012) and the remaining items were created by the researchers based on a survey that was used in debugging and robotics studies conducted in the past. Scores in each category reflected participants' beliefs in that specific category and the sum of the scores from the three categories generated a total self-efficacy score.

Procedures

Session 1 – Debugging Intervention

Each student was assigned to one of the programming language groups (Coding Language or Narratives). Each group received a different debugging worksheet written in either coding language or narratives, but had the same maze problem. To solve the maze problem on the worksheet, participants were asked to find the errors in the maze directions. Once they found the errors, they wrote revised directions on the worksheet through code or narratives, depending on the language group they were assigned.

Once revised directions were written, participants tested the revised directions through two types of embodiment (Full or Low embodiment). The Full embodiment students checked their revised directions on the floor maze by moving their body while holding the worksheet, whereas the Low embodiment group tested the revised directions by moving a paper character on the worksheet.

Session 2 – Post-Test

After the debugging intervention, the children responded to four post-tests (debugging skills, programming test, self-report survey, and a TOPs-3 interview) to investigate the effect of cognitive skills related to programming.

Results

The results from a one-way analysis of variance (ANOVA) showed that there was no significant differences on grade distribution within four groups, F (1, 55) = 0.226, p = .878. To analyze post-test data, a two-way ANOVA was used, accompanied with a Mann Whitney test. Analyses of the data examined differences between groups in students' debugging and programming skills, cognitive skills related to CT, and awareness about self.

This section is organized into four sections (debugging competencies, programming skills including programming performance and programming efficiency, cognitive skills, and self-efficacy) to report the comparison among four intervention groups combined with embodiment and programming language factors.

Programming Debugging Test

The debugging post-test measured sequence, pattern recognition (repeat), and conditional, with scores ranging from 0 to 20. When comparing the main effect of embodiment, there is a significant interaction effect, F (1, 55) = 4.611, p = .036, η^2=.077, indicating that the effect of embodiment on debugging depends on the language type (Table 8). When looking at the four intervention groups, those in the Full embodiment with Coding language group (M = 16.375, SD = 3.667*)* performed the best in debugging performance, followed by Low embodiment with Narratives, Low embodiment with Coding language, and, lastly, Full embodiment with Narratives (Figure 8). However, there was no significant difference within each factor.

Table 8

Two-way ANOVA on Debugging Test Scores by Embodiment and Programming Language Factor

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Embodiment Condition	32.198	1	32.198	1.884	0.175	0.033
Language Condition	34.034	1	34.034	1.991	0.164	0.035
Embodiment*Language	78.816	1	78.816	4.611	0.036*	0.077
Error	940.047	55	17.092			

$R^2 = .137$ (Adjusted $R^2 = .090$), * $p < .05$,
Note. Maximum score is 20.

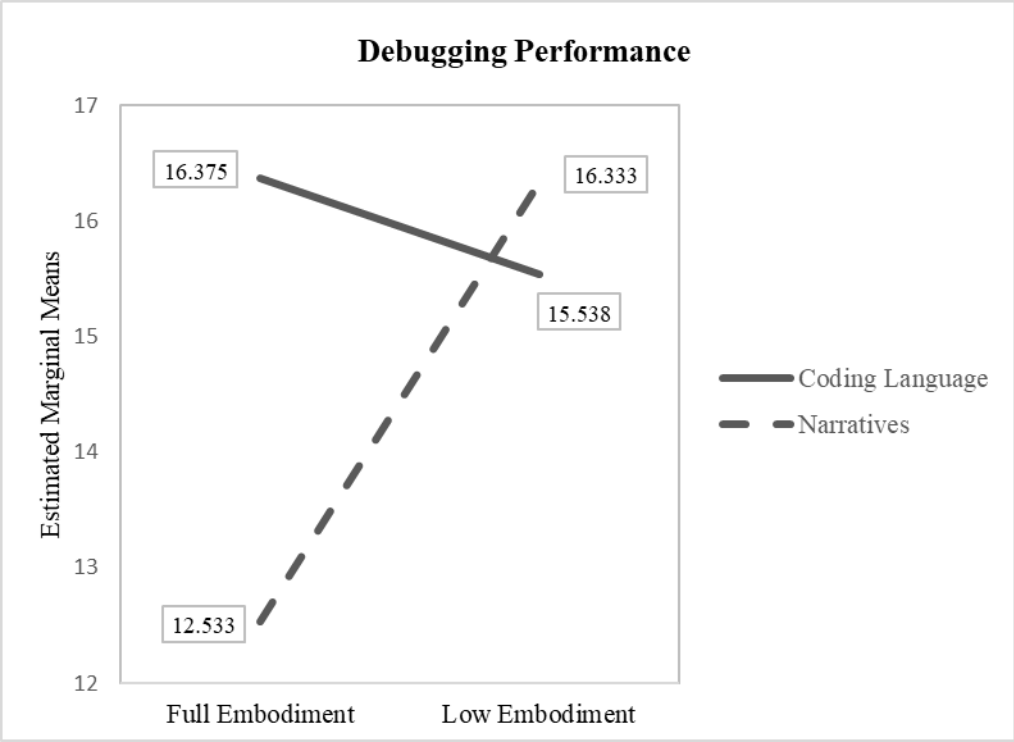


Figure 8. A profile plot of debugging test scores among the four groups

Impact of Debugging Intervention on Programming Performance

Programming Performance. Participants' performance on the programming task was analyzed by looking at how many levels a participant accomplished using the Tynker iPad programming software. There was no significant main effect and no interaction between the two factors − embodiment condition, F (1, 55) = .691, p = .409; language condition, F (1, 55) = .760, p = .387; interactions, F (1, 55) = .004, p = .949.

Programming Efficiency. Programming efficiency was calculated by adding up the stars each participant received on each programming game level (scores from 0 to 63), which indicates the nobleness strategy used during play. There was no significant impact on the degree of embodiment, F (1, 52) = 2.584, p = .114, or type of programming language, F (1, 52) = 0.105, p = .74. Also, there was no interaction effect, F (1, 52) = 0.435, p = .512.

TOPs-3 Cognitive Skill Interview

To compare cognitive skills, the sum of the average of each cognitive skill subcategory (sequence, problem solving, predicting, and determining cause) from the Elementary Test of Problem Solving (TOPs-3) was calculated. There was a significant score difference in total cognitive skills within the embodiment condition, F (1, 55) = 4.615, p = .036, η^2= .077 (Table 9). The Low embodiment group (M = 5.264, SD = 1.149) performed significantly better on cognitive skills than the Full embodiment group (M = 4.549, SD = 1.319), with a mean difference of 0.715 (Table 10). No interaction between the two factors was found, F (1, 55) = 0.029, p = .865.

In terms of programming language type, no significant differences between groups were found. When comparing the four groups, the Low embodiment with Narratives group scored the highest while the Full embodiment with Coding language group scored the lowest on cognitive skills.

Table 9

Two-way ANOVA on TOPs-3 Interview Scores by Embodiment and Programming Language Factor

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Embodiment Condition	7.341	1	7.341	4.615	0.036*	0.077
Language Condition	0.385	1	0.385	0.242	0.625	0.004
Embodiment*Language	0.047	1	0.047	0.029	0.865	0.001
Error	87.488	55	1.591			

R^2 = .083 (Adjusted R^2 = .033), * $p < .05$

Table 10

Mean and Standard Deviation of TOPs-3 Interview Scores by Group

	Coding Language		Narratives		Total	
	M	*SD*	*M*	*SD*	*M*	*SD*
Full Embodiment	4.443	1.297	4.662	1.379	4.549	1.319
Low Embodiment	5.207	1.257	5.313	1.089	5.264	1.149
Total	4.786	1.315	4.987	1.265	4.887	1.282

Note. Maximum score is 8. (Sum of TOPs Average on Sequence, Problem Solving, Predicting, and Determining Cause)

TOPs-3 Cognitive Skill Subcategories: Problem Solving. For a more precise analysis, the four subcategories that make up cognitive skills (sequence, problem solving, predicting, and determining cause) were examined separately. The average score, ranging from 0 to 2, in each category was used for this analysis. Within the embodiment condition, obvious patterns were found in which the Low embodiment group showed better performance than the Full embodiment group across all four subcategories. In particular, there was a significant difference only in problem solving between the Low embodiment group (M = 1.320, SD = 0.425) and the Full embodiment group (M = 1.030, SD = 0.437), mean difference 0.29, F (1, 55) = 6.306, p = .015, η^2= .079 (Figure 9). However, there were no significant results in the programming language condition.

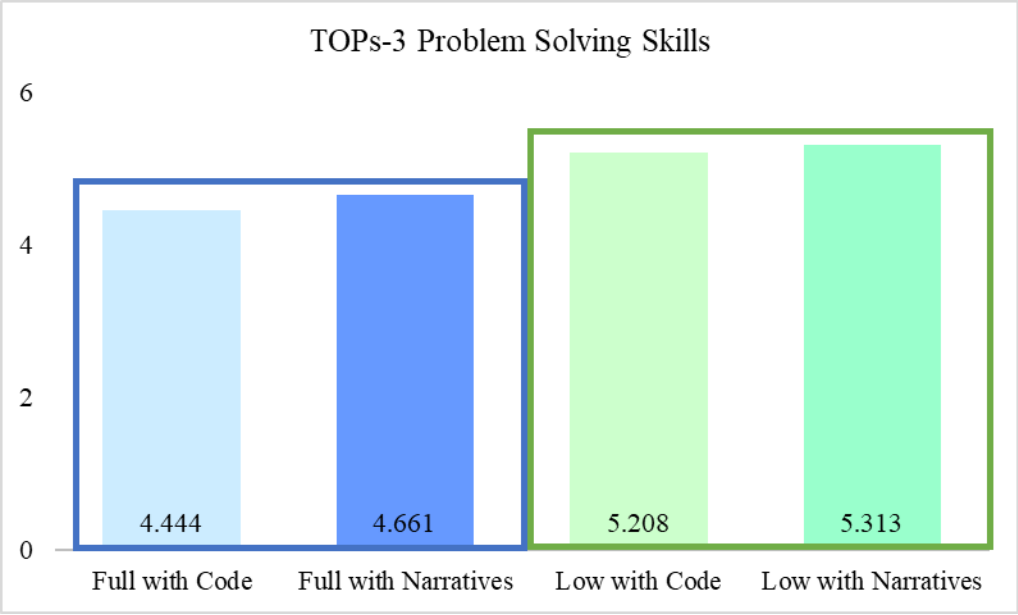


Figure 9. Mean of TOPs-3 subcategories: problem solving scores

Self-Efficacy Survey

Participants' attitude towards self-efficacy was assessed to measure whether certain groups had significantly higher self-efficacy, specifically competence or interest in coding, computer, and behavior in the classroom, after the debugging intervention (scores from 0 to 75). Significant differences were found within the two factors. In the embodiment condition, participants in the Low embodiment group (M = 43.64, SD = *6.701*) reported significantly higher level of self-efficacy than the Full embodiment group (M = 38.71, SD = 5.900), F (1, 52) = 8.712, p = .005, η^2= .137. Under the programming language condition, the Narratives group (M = 42.77, *SD = 6.268*) was significantly higher than the Coding language group (M = 39.28, SD = 6.803), F (1, 52) = 4.269, p = .044, η^2=.072 (Table 11 & 12). Those who employed Low embodiment with Narratives reported the highest self-efficacy whereas those who used Full embodiment with Coding language reported the lowest self-efficacy. However, no interaction effect between the two factors was found.

Table 11

Two-way ANOVA on Self-Efficacy Scores by Embodiment and Programming Language Factor

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Embodiment Condition	328.830	1	328.830	8.712	0.005**	0.137
Language Condition	161.133	1	161.133	4.269	0.044*	0.072
Embodiment*Language	25.996	1	25.996	0.689	0.410	0.012
Error	2076.026	52	37.746			

R^2=.206 (Adjusted R^2=.163), * p < .05, **p < .01

Table 12

Mean and Standard Deviation of Self-Efficacy Survey Scores by Group

	Coding Language		Narratives		Total	
	M	*SD*	*M*	*SD*	*M*	*SD*
Full Embodiment	37.75	5.639	39.73	6.193	38.71	5.900
Low Embodiment	41.15	7.830	45.80	4.828	43.64	6.701
Total	39.28	6.803	42.77	6.268	41.05	6.714

Note. Maximum score is 55 (11 questions using 5-point Likert scale).

Self-Efficacy Subcategories. Analysis of each self-efficacy subcategory shows significant difference on coding confidence and classroom behavior within factors. In the embodiment condition, participants in the Low embodiment group reported a higher level of self-efficacy than the Full embodiment group both in coding competences, F (1, 55) = 7.082, p = .010, η^2=.114, and classroom behavior, F (1, 55) = 5.257, p = .026, η^2=.087. Also, the Narratives group was significantly higher than the Coding language group in the coding confidence, F (1, 55) = 4.758, p = .033, η^2=.080.

Discussion

In the second study, participants utilized different degrees of embodiment (Full vs. Low embodiment) and programming language types (Coding language vs. Narratives) during the debugging intervention, which directed young students to debug given coding scripts on a worksheet which contained errors they needed to fix. Full embodiment refers to learners moving their body to enact debugged code, while low embodiment is defined as learners manipulating a paper character to simulate revised code. The researcher was

interested in designing instructional practices that cultivate computational thinking and gain insight into the learning and impact of debugging practices toward self-efficacy. Thus, the research conducted post-tests to see the impact of debugging practices toward CT, cognitive skills related to CT (e.g., problem solving, sequence, prediction, and determining cause), and self-efficacy.

The results suggest that compared to full embodiment, low embodiment tends to have better support on young students' learning in cognitive skills related to CT. This confirms the research hypothesis regard to cognitive skills. However, Low embodiment also contributes to a higher attitude towards belief of one's competences (e.g., self-efficacy), in contrast to what was hypothesized. This may be attributed to the fact that debugging is a highly complex and dynamic activity that requires considerable cognitive capabilities. Thus, less cognitive load involved with low embodiment may have helped children to better outcomes in this study. According to Chandler and Sweller (1996), who explained the relationship with cognitive load and content learning, if learners' extraneous cognitive load is overloaded, they may have difficulty perceiving and understanding the learning content. Therefore, low embodiment allowed the participants to easily associate a command block with resulting outcomes, which made dealing with the coding errors more manageable as they used their hands to move the character according to the programming script.

Results from the type of programming language suggest that narratives integrated into the debugging intervention empowers young students to have high self-efficacy. These findings are closely related to what is already known about programming language: novice programmers improve their first programming experience with

programming language that resembles natural everyday language, such that reducing mechanical difficulties of programming syntax improve novice programmers' first programming experience (Kelleher & Pausch, 2005). Although the coding language group also used words students use every day (i.e., forward, turn left, turn right), this presented a word itself, not a sentence, on a flow chart. However, for the novice programmer, dealing with symbol representation of a flow chart and making connections between words can be overwhelming and discouraging. These findings underscore the need to create instructions that make it easy to get started with programming.

When comparing the four intervention groups, a statistically significant difference was found on debugging skills. Those in the Full embodiment with Coding language group performed the best in debugging performance, followed by those who used Low embodiment with Narratives, Low embodiment with Coding, and, lastly, Full embodiment with Narratives. Specifically, debugging performance changes within the Full embodiment condition was extremely dramatic, while the two programming language groups across low embodiment showed similar performance. Within full embodiment, debugging performance scores lowered from coding language to narratives. One explanation for this can be found from the perspective of "desirable difficulties" (Bjork, 1994; Bjork & Bjork, 2011; Bjork & Linn, 2006), meaning that some degree of difficulty is necessary in learning environments and that unchallenging instructional tasks can limit high-level thinking learning. Another explanation can be nature of programming language type. Even though the narratives type of programming language was manageable for them, there was limited room to develop CT skills meaningfully. On the other hand, using coding language is more abstract than narratives, and the seemingly

demanding nature of a coding language offered the opportunity to develop a deeper understanding of the concept.

In particular, it was not conclusive that the type of text-based programming language, along with the two embodiment activities, affected learners' debugging and programming skills. One possible explanation for the inconclusive results is that the text-based programming language was not developmentally appropriate for participants in 2nd to 3rd grade, whereas, most learners' coding software available for touch-based tablets utilize a graphical block-based (e.g., constructing programs using graphical objects, typically in a drag-and-drop interface) programming language (e.g., Scratch Jr.),which is developmentally appropriate. Another explanation can be explained by the limitation of the study in that test items on the debugging test used a similar format as the coding language, which could lead bias results. These findings inspired the design of the next study to investigate desirable difficulties with congruent and incongruent gestures.

This study had other limitations as well. It is difficult to examine the acquisition of cognitive skills using only a post-test only research design. Also, using electronic test possibly reduced the appropriateness of test an instrument. Compared to the paper-based debugging test which found a significant difference between groups, features in the iPad Tynker software, such as immediate feedback for coding script, trial and errors, and hints, reduced the discrimination power of the programming test.

The results from the two preliminary studies (Study 1 & Study 2) also suggest that the use of low embodiment tends to better support learners' learning of CT skills and attitude towards belief of one's competencies compared to full embodiment. Relatively low cognitive load associated with low embodiment led to better CT learning and self-

efficacy. Also, conditional similarity (the third person view where a student watched the character's movement while manipulating the character on paper followed by a programming script) between iPad usage and low embodiment possibly yielded more favorable outcomes for the Low embodiment group. Study 3 re-evaluated the effects of low embodiment through cognitive load and conditional similarity with a touch-based tablet programming environment (e.g., perspective taking) by employing direct and surrogate embodiment.

V – STUDY THREE

Dissertation Study: Using Gestures and Embodiment for Debugging Activities

This chapter details the purpose and methods of Study 3. This chapter is divided into six parts. The first section describes the background of the study related to the two preliminary studies. The second part addresses research questions and the hypothesis. The third section describes the participants and the overall research design. The fourth section focuses on the instruments of the study. The fifth section details the procedure. The final section describes data analysis procedures corresponding to the study's research questions and discusses the findings.

Overview

As an extension of the two preliminary studies, Study 3 offers another embodied instructional approach ─ gesture embodiment (i.e., congruent and incongruent gesture) ─ coupled with direct embodiment and surrogate embodiment to teach computational thinking (CT) and foster learners' self-efficacy and persistence while debugging programming scripts. The embodied debugging activities in this study provide guidelines for practicing CT skills without digital devices.

The two previous studies examined developing learners' (kindergarten to 3rd graders) debugging and programming skills, cognitive skills related to CT, and self-efficacy. Study 1 implemented different types of embodiment (full embodiment vs. low

embodiment). Based on the first study, a follow-up study also included type of text-based programming language. The main finding from the two studies suggests the effectiveness of low embodiment, where learners perceptually simulate programming scripts to an external agent using their hands, rather than full embodiment (e.g., a student physically enacts programming scripts to debug the programming code). In addition, a narrative type of programming language promotes a high level of self-efficacy compared to a coding language. The findings and limitations from the preliminary studies provided insight into Study 3 to integrate different types of low embodiment and gesture embodiment. Thus, this study applied a graphic-based block programming language during unplugged debugging practice to examine students' development of computational thinking, preparation for text-based programming, and self-efficacy.

Gesture Embodiment: Congruent and Incongruent Gesture

Many programming applications for touch-based tablets are designed to drag and drop coding blocks with a top-down or left-right order, which does not correspond to the movement of a programming character. In my observations in an after-school coding classroom, many young students who are relatively new to programming tend to place coding blocks in the same direction that the programming character would move. For example, when students arrange a code for a character to move in the maze, they place the coding block in the same direction as the moving path in the maze, rather than arrange it in a top-down or left-right order, unless the teacher instructs them to do so. These observations are reminiscent of research on perceptual congruency versus incongruency in learning in terms of how students arrange paper coding blocks on a

worksheet during an intervention (path way order vs. left-right order). Congruent gesture refers to placing coding blocks in the same direction that the programming character will move (gesture is congruent to perception), whereas incongruent gesture refers to a pattern which is placed in a top-down or left-right order (gesture is not congruent to perception).

Research Questions and Hypotheses

Study 3 examined whether an embodied debugging experience (e.g., congruent gesture vs. incongruent gesture, direct embodiment vs. surrogate embodiment) with graphic-based block programming language contributes to young students' development of computational thinking, self-efficacy, and persistence. Furthermore, the study investigated whether experience in a graphic-based block programming language during the early grades can predict text-based programming competencies (e.g., syntax programming language such as Python) in the future. This study investigated the following research questions:

Research Question 1: Do gestures and embodiments for debugging activities impact students' learning of programing, self-efficacy, and persistence?

Hypothesis 1: Learners will more likely show better programming skills, self-efficacy, and persistence after debugging activities through gestures and embodiments.

Research Question 2: Which type of activity out of the four experimental groups, created by combining two gestures and two embodiments, led to better performance in programming and personal development?

Hypothesis 2:

1) Learners who participate using Incongruent gesture with Surrogate embodiment will show the greatest graphic-based block programming performance than those in the other three groups.
2) Those who use Incongruent gesture with Surrogate embodiment will more likely perform better in text-based block programming.
3) Those who use Congruent gesture and Surrogate embodiment will show higher levels of self-efficacy.
4) Learners who use Incongruent gesture with Direct embodiment will show a greater degree of persistence.

Research Question 3: How do different hand gestures to arrange coding blocks (congruent vs. incongruent gesture) affect learners':

1) Performance in graphic-based block programming?
2) Performance in text-based block programming (to examine knowledge transfer)?
3) Perception of self-efficacy?
4) Persistence in the face of difficulties?

Hypothesis 3: Learners in the Congruent gesture group will exhibit higher self-efficacy than those in the Incongruent group, while those in the Incongruent gesture group will more likely show better graphic-based programming, text-based programming, and persistence.

Research Question 4: How do the different types of embodiment (Direct vs. Surrogate embodiment) affect learners' performance in the areas of programming competencies

(graphic-based block programming, text-based block programming), self-efficacy, and persistence?

Hypothesis 4: As discovered in Studies 1 and 2, the Surrogate embodiment group will demonstrate greater performance across all outcomes than the Direct embodiment group, except on persistence.

Research Question 5: Is there a relationship between:

1) Graphic-based block programming and text-based block programming?
2) Self-efficacy and graphic-based and text-based programming?
3) Persistence and graphic-based and text-based programming?

Hypothesis 5:

1) Learners who demonstrate better skills with graphic-based block programming will show higher achievement on text-based block programming.
2) Learners with higher self-efficacy scores will show higher achievement on both graphic-based and text-based block programming.
3) Learners with better persistence scores will show higher achievement on both graphic-based and text-based block programming.

Participants

The study was conducted with 84 2nd to 3rd grade students (56 males, 28 females) who participated in an after-school coding program or a coding summer camp in a New York City public school. Participants included 50% second graders and 50% third

graders. The ethnicity of the participants consisted of about 50% African American, 29.8% Caucasian, 11.9% Hispanic, and 8.3% Asian.

Research Design

The study was designed as a 2 x 2 factorial experiment (Table 13) with two main factors − type of gesture and type of embodiment. Participants were randomly assigned to one of four conditions (Table 14):

Condition 1: Congruent Gesture with Direct Embodiment

Condition 2: Congruent Gesture with Surrogate Embodiment

Condition 3: Incongruent Gesture with Direct Embodiment

Condition 4: Incongruent Gesture with Surrogate Embodiment

Table 13

2 x 2 Factorial Experimental Groups

Type of Gesture	Type of Embodiment	
	Direct Embodiment	Surrogate Embodiment
Congruent Gesture	Group 1: Congruent Gesture with Direct Embodiment (N = 23)	Group 2: Congruent Gesture with Surrogate Embodiment (N = 27)
Incongruent Gesture	Group 3: Incongruent Gesture with Direct Embodiment (N = 24)	Group 4: Incongruent Gesture with Direct Embodiment (N = 25)

Table 14

Intervention Descriptions

Types of Gestures	Type of Embodiment	
	Direct Embodiment	Surrogate Embodiment
Congruent Gesture	Congruent Gesture with Direct Embodiment: During the unplugged debugging intervention, students find errors and revise coding blocks in the maze path on a worksheet. They test the revised code by manipulating a character on the worksheet maze while doing a verbalization of the code.	Congruent Gesture with Surrogate Embodiment: During the unplugged debugging intervention, students find errors and revise coding blocks in the maze path on the worksheet. They test the revised code by giving verbal commands to a researcher who manipulate a character on the worksheet maze.
Incongruent Gesture	Incongruent Gesture with Direct Embodiment: During the unplugged debugging intervention, students find errors and revise coding blocks on the worksheet through left to right order. They test the revised code by manipulating a character on the worksheet maze while doing a verbalization of the code.	Incongruent Gesture with Direct Embodiment: During the unplugged debugging intervention, students find errors and revise coding blocks on the worksheet from left to right order. They test the revised code by giving verbal commands to a researcher who manipulates a character on the worksheet maze.

This study examined the effect of two main factors during debugging intervention − type of hand gestures and type of embodiment − on CT skills, self-efficacy, and persistence. The intervention was unplugged debugging activities (e.g., not using any digital devices) of given programming code with errors on a worksheet, followed by embodied activities. Before the debugging intervention, pre-tests (i.e., graphic-based block programming test, self-efficacy survey) were administered to the students. Following the unplugged debugging activities, post-tests were administered.

Type of Gesture: Congruent vs. Incongruent Gesture to Arrange Paper Coding Blocks

One variable in the study is the arrangement of the coding blocks using different gestures. The gesture variable was applied when learners debug and revise the given coding script on a worksheet using paper coding blocks. Under the gesture variable, there are two conditions: perceptual congruency with gesture (congruent gesture) and perceptual incongruency with gesture (incongruent gesture) (Figure 10). Congruent gesture refers to when the arrangement of the paper coding blocks on the worksheet is the same as the movement of a character according to the coding script. Incongruent gesture is when the arrangement of the paper coding blocks on the worksheet is not the same as the movement of an external surrogate (e.g., place coding blocks left to right regardless of the path of the sprite's movement).

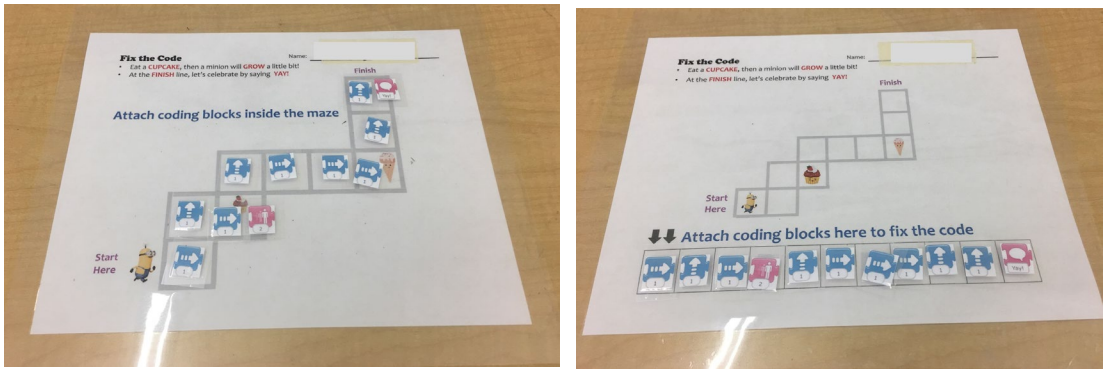


Figure 10. Congruent gesture and incongruent gesture worksheets

Type of Embodiment: Direct and Surrogate Embodiment to Examine Debugged Code

This debugging study incorporated a different degree of embodiment, namely, direct embodiment and surrogate embodiment. Immediately after students debug and revise a given code on the worksheet using different types of gestures, they test a revised code by implementing one of the embodiment types.

Under the Direct embodiment condition, students tested revised code by moving a character on the worksheet (Figure 11). In the Surrogate embodiment condition, students examined their revised code by making verbal commands to the researcher (Figure 12). The researcher played the role of an external surrogate to the students by following the commands and moving a character on the worksheet maze. Therefore, students in the Direct embodiment group utilized part of their bodies (i.e., hands) to enact a paper character followed the revised coding script). In contrast, students in the Surrogate embodiment group only gave verbal commands to the researcher (i.e., surrogate object) and observed the researcher's movement. Additionally, two embodiment types demonstrated and provided feedback on how the students' coding worked from different perspectives (e.g., direct embodiment as a first-person perspective versus surrogate embodiment as a third-person perspective). Therefore, receiving feedback from these two embodiment types also served as different forms of perspective taking.

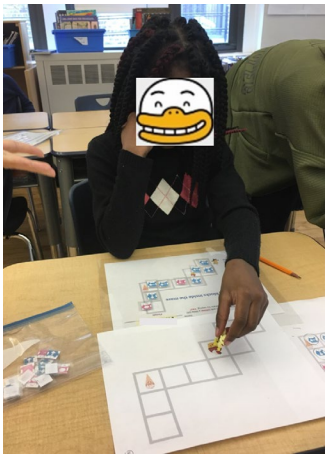


Figure 11. An activity of Direct embodiment in which students enact a character

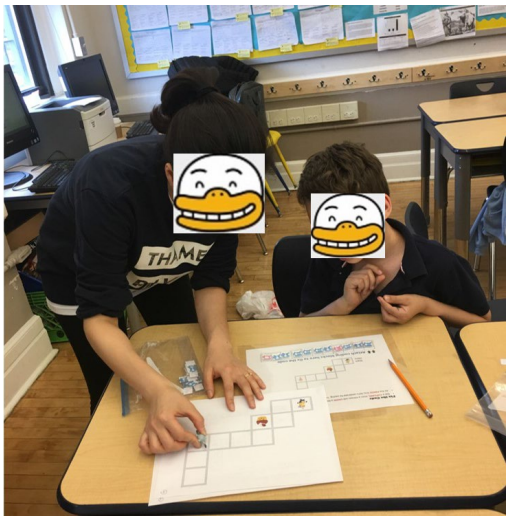


Figure 12. A picture of Surrogate embodiment showing students making verbal commands to the researcher

Materials

Three dependent variables were measured during pre- and post-measures to evaluate participants' understanding of graphic-based programming and perception on

self-efficacy. The post-test measured two additional dependent variables to evaluate understanding of text-based programming and level of persistence.

Pre- and post-measures:

- Graphic-based block programming paper test
- Self-efficacy survey

Post-measures only:

- Text-based block programming paper test (Transfer Test)
- Persistence task: Challenge task (using Kodable iPad programming app)

Graphic-Based Block Programming Test

The content measured in this block-based test were concepts and skills that are fundamental to computer programming and computational thinking. To measure students' graphic-block based programming skills, a paper-based programming skill test was developed (Appendix D). The test items were based on Bloom's taxonomy, which Meerbaum-Salant, Armoni, and Ben-Ari (2013) applied in the context of computer science education.

1) Understanding: The ability to summarize, explain, exemplify, classify, and compare computer science concepts, including programming constructs;

2) Applying: The ability to execute programs or algorithms, to track them, and to recognize their goals;

3) Creating: The ability to plan and produce programs or algorithms.

Questions were derived from essential programming strategies such as sequence, pattern recognition (repeat), and debugging. The test consisted of five items of fill-in-the-blank

and multiple choice items. The maximum score was 9 points. Question 1 measures understanding of programming, comparing programming outcomes among three multiple choices. Questions 2, 3, and 4 examine application ability by executing programming, recognizing the expected outcome, and creating more effective programming through abstraction. Question 5 measures creation ability of programming, as students go through a programming sequence to achieve a goal, detect patterns, and produce efficient programming. The time it took for the participants to complete the test was not recorded; however, it was observed that participants spent approximately less than 10 minutes taking the test.

Text-Based Block Programming Test

The purpose of this transfer test is to measure whether the embodied debugging intervention is helpful in preparing for text-based programming (i.e., Python) in which learners will learn in a higher grade. The findings from this test will guide how to structure instruction more effectively to transfer CT skills to later learning of a different programming language. Similar to the graphic-based block programming test, this test was developed based on Bloom's taxonomy. The questions were derived from essential programming strategies such as sequence, pattern recognition (repeat), and debugging. The test consisted of five items of fill-in-the-blank and multiple choice items. The transfer test was conducted as a post-test. Questions were mostly multiple choice as learners would not be familiar with a new programming language.

Questions 1, 3, and 5 measure both understanding and application ability of programming, which is related to summarize, explain, compare programming constructs,

execute programming, and recognize the expected outcome (Appendix F). Questions 2 and 4 assess both application and creation ability of programming by execute programming, recognize the expected outcome, and revise programming sequence to achieve a goal and produce efficient programming.

Self-Efficacy Survey

To measure participants' self-efficacy on perceived competence in programming and digital device usage, a self-efficacy survey was administered before and after the intervention. The four questions were based on 5-point Likert scale (Appendix E). Similar to the survey adopted in the preliminary studies, the items were derived from the Positive Technology Development Questionnaire (Bers et al., 2012). The sum of the scores from the two categories generates a total self-efficacy score.

Persistence Task

An electronic persistent task was administered on an iPad using an advanced level game in the Kodable programming application as a post-test. Learners were asked to try their best to solve the challenge task within 20 minutes. If they completed the problem, the researcher recorded the time they spent. However, if they wanted to leave the game before 20 minutes, students were given two options, try again or move on to science simulations. The time it took for them to complete or give up on the challenge task was used as a proxy for persistence. The playing time (seconds) until the student stopped was recorded for analysis.

Procedure

The study was conducted in four sessions, 50 minutes each, at New York City public schools during after-school and summer camps.

Session 1 – Pre-Test

Participants in the study completed pre-tests and explored a coding application in the first session. In order to make sure students from different schools had no significant background knowledge in the measured outcomes (i.e., computational thinking skills and self-efficacy), the study used a pre- and post-test between subjects design. A graphic-based block programming application, Scratch Jr, was introduced to allow students to freely explore a coding application, as it seemed new to some students. To ensure the pure impact of the intervention, the researchers did not provide any tutorial on the coding application or coding concepts. Participants then completed a graphic-based block programming test and a self-efficacy survey.

Session 2 – Debugging Intervention

In the second session, the learners worked through debugging interventions according to the treatment group they were assigned. The task in the debugging intervention was to find and revise given coding script errors in order to successfully solve a mission. First, students practiced under the gesture condition. They arranged graphic-based paper coding blocks on one of two different worksheets (either congruent gesture or incongruent gesture) to fix or revise the given code. Students in the Congruent

gesture group detached and replaced Velcro coding blocks onto the worksheet, which was compatible to the maze path in which a character would move according to the coding script the student made. On the other hand, the Incongruent group debugged with Velcro coding blocks from left to right order, which is incongruent with the maze path. The paper coding blocks were made of a laminated paper with Velcro, which enabled students to easily attach or detach coding blocks on the worksheet.

Immediately after revising the worksheet, students completed one of the embodiment activities. By having the revised code on the worksheet, students physically enacted (direct embodiment) or commanded the surrogate to enact the revised code depending on their embodiment group (surrogate embodiment). Specifically, the Direct embodiment group tested the revised code by manipulating a character on the worksheet maze while verbalizing the code. In the Surrogate embodiment group, students tested the revised code by giving verbal commands to the researcher, who manipulated a character on the worksheet maze. This was an iterative process until the end point of the given mission was reached. The following is a summary of the four intervention groups, made up of two factors of gesture and embodiment.

Session 3 – Post-Test 1

After successful completion of the debugging intervention through gesture and embodiment, all participants completed the post-tests. They completed the tests that were administered during the pre-test − a graphic-based block programming test and a self-efficacy survey.

Session 4 – Post-Test 2

Two other post-tests were presented to students − text-based block programming and a persistence task. These tests were assessed as post-tests only.

Results

Descriptive and inferential statistical analyses were used to answer the effect of hand gesture when arranging graphic-based coding blocks and the practice of body movement during unplugged debugging activities. The outcomes in the study measured two domains: 1) computational thinking skills (i.e., graphic and text programming skills) and 2) personal attributes (i.e., self-efficacy, persistence). In total, 84 2nd and 3rd grade students participated in the study. In order to check even distribution of participants' grade in each experimental group, Multinominal Logistic Regression was performed The results showed that there was no statistically significant difference of grade by group type (p = .883), confirming grades were evenly distributed between each group.

In this study, learning outcomes from the debugging intervention were measured with two tests, a graphic programming test and self-efficacy survey. To measure learning outcomes on both tests, the test scores on the pre- and post-tests were used. Using a paired sample t-test, learning outcomes from the graphic programming test and self-efficacy survey were compared across all participants regardless of the experimental group they were assigned. To compare learning outcomes between all four experimental groups, an Analysis of Variance (ANOVA) test was conducted as well as the pairwise comparisons. The impact of the two factors in regards to the participants' learning

outcomes on the pre- and post-tests was also examined using a 2x2 ANOVA by comparing the gesture factor, embodiment factor, and the combination of the two factors.

Graphic-Based Block Programming Test

Participants were randomly assigned to one of four conditions. The results of the one-way ANOVA test on the graphic programming pre-test confirmed that the four experimental groups were not significantly different in terms of grade and graphic programming skills, F (3, 73) = 1.566, p = .205.

When looking at learning outcomes from the graphic programming test, there was a significant difference between pre- and post-test scores across the four experimental groups, when the two factors were excluded in the analysis. This suggests that learning from graphic programming occurred after the debugging intervention. However, no significant learning was found among the four experimental groups or from the two factors.

In the post-test analysis, a 2x2 ANCOVA analysis was conducted. Among the three concepts measured in graphic programming (pattern recognition, sequence, and debugging), only sequence skills on the graphic programming test were significantly different between the two gesture groups. As shown in Table 15, sequence performance yielded a significant difference for the gesture factors between congruent gesture and incongruent gesture, F (1, 73) = 4.200, p = .044, η^2= .054. Students who used incongruent gesture scored higher than those who used congruent gesture, with a mean difference of 0.22 (Figure 13). Embodiment was not a significant factor, F (1, 73) = .373, p = .543, η^2= .002, and also had no impact on the interaction between the two, F (1, 73) =

2.346, $p = .130$, $\eta^2 = .013$. There was no significant impact of the embodiment factor and no interaction between the two factors on sequence.

Table 15

Two-way ANCOVA on Graphic Programming Post-Test, Sequence Scores by Gesture and Embodiment

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Gesture	19.456	1	19.456	4.200	0.044*	0.054
Embodiment	1.729	1	1.729	0.373	0.543	0.002
Gesture*Embodiment	10.867	1	10.867	2.346	0.130	0.013
Error	338.154	73	4.632			

$R^2 = .088$ (Adj. $R^2 = .051$), * $p < .05$

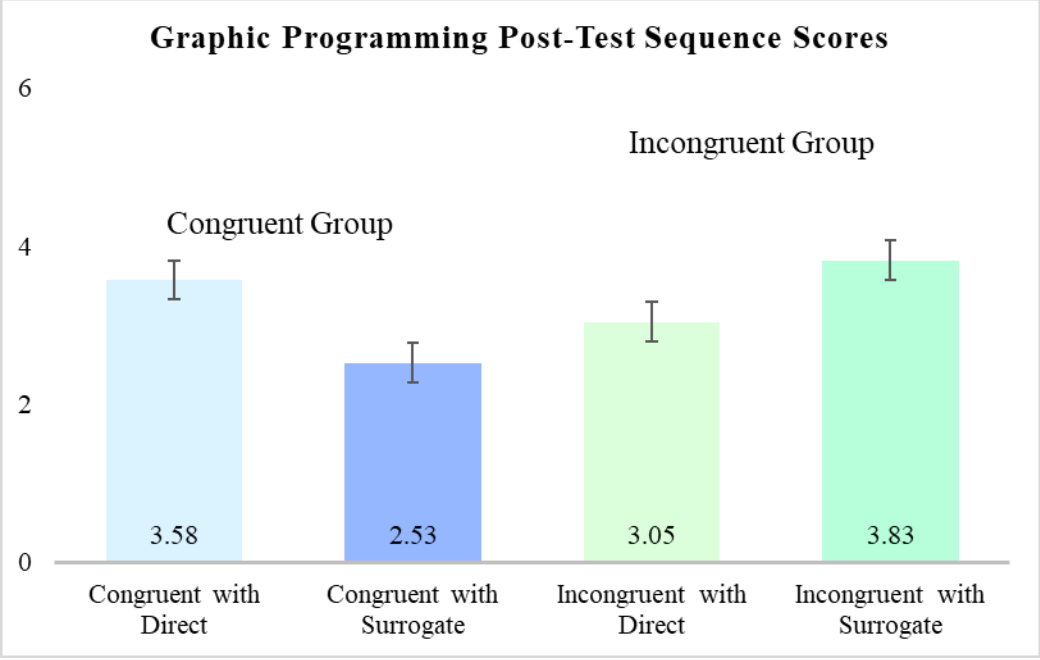


Figure 13. Graphic programming post-test sequence scores

There was no statistical significance found in programming total scores combining all questions from pattern recognition, sequence, and debugging concepts. Meanwhile, within the gesture group, the Incongruent group showed higher scores (M = 5.74, SD = 2.643) than the Congruent groups (M = 4.45, SD = 3.064) on the graphic programming test, with an estimated mean difference of 1.29. (Table 16). When comparing the four groups, those in the Incongruent gesture with Surrogate embodiment group had the highest scores on the graphic programming test (M = 6.19, SD = 2.750), whereas the Congruent gesture with Surrogate embodiment group achieved the lowest scores (M = 3.84, SD = 3.096). Again, this reports a trend but there was no statistical significance found among the four groups.

Table 16

Mean and Standard Deviation of Graphic Programming Post-Test Scores by Group

	Direct Embodiment		Surrogate Embodiment		Total	
	M	*SD*	*M*	*SD*	*M*	*SD*
Congruent Gesture	5.05	2.990	3.84	3.096	4.45	3.064
Incongruent Gesture	5.22	2.487	6.19	2.750	5.74	2.643
Total	5.14	2.720	5.07	3.116	5.10	2.914

* $p < .05$, *Note*. Maximum score is 9.

These results address the following hypotheses.

H1. Participants will learn graphic programming test after debugging activities through gestures and embodiments.

Although graphic programming learning occurred between the pre- and post-tests across the four experimental groups, no significant differences were found among the four intervention groups and between the two factors.

H2.1. Participants in the Incongruent gesture with Surrogate embodiment group will show the greatest graphic-based block programming performance than the other three groups.

Although students in the Incongruent gesture with Surrogate embodiment group received the highest score on the graphic-based block programming test, no significant differences were found among the four intervention groups.

H3. Participants in the Incongruent gesture group will exhibit a higher level of performance on graphic-based block programming.

The results confirm that students who practiced debugging with incongruent gesture scored significantly higher than those who used congruent gesture.

H4. The Surrogate embodiment group will demonstrate greater performance across all outcomes compared to the Direct embodiment group, except on persistence.

The results show that there were no significant differences between the Direct embodiment and Surrogate embodiment groups.

Text-Based Block Programming Test

The text-based block programming test was administered to determine participants could transfer what they learned in the embodied debugging activities by utilizing a graphic-based coding block. It assessed three concepts related to computational thinking: 1) pattern recognition, 2) sequence, and 3) debugging. The participants had no experience with text programming during the debugging intervention; therefore, this required transfer and retention ability. This learning outcome was a paper-based post-test with five items. We hypothesized students in the Incongruent gesture with Surrogate embodiment group would most preferably perform the transfer test with text-based block programming.

From a 2x2 ANOVA analysis, the results show there was a significant difference on pattern recognition in text-based block programming within the embodiment factor F (1, 70) = 1.130, p = .030, η^2= .065 (Table 17). Sequence and debugging skills in text programming test did not show any significance. Due to the binominal nature of the pattern recognition item on the test, a binary logistic regression was conducted along with ANOVA. The binary logistic regression also confirmed the significant difference within the embodiment groups in performing pattern recognition, (B = -1.073, SE = .494, $Wald$ = 4.720, p = .030). The estimated marginal mean indicates that the Surrogate embodiment group outperformed the Direct embodiment group on pattern recognition (mean difference = .248, Figure 14). This result does not correspond to the result of the graphic-block based programming test, which discovered significance within the gesture groups. Further interpretation will be discussed in the discussion section.

There was no significant impact of the embodiment factor and no interaction between the two factors on sequence.

Table 17

Two-way ANOVA on Text-based Block Programming, Sequence Scores by Gesture and Embodiment

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Gesture	0.065	1	0.065	0.282	0.597	0.004
Embodiment	1.130	1	1.130	4.898	0.030*	0.065
Gesture*Embodiment	0.636	1	0.636	2.754	0.101	0.038
Error	16.154	70	0.231			

$R^2 = .103$ (Adj. $R^2 = .065$), * $p < .05$

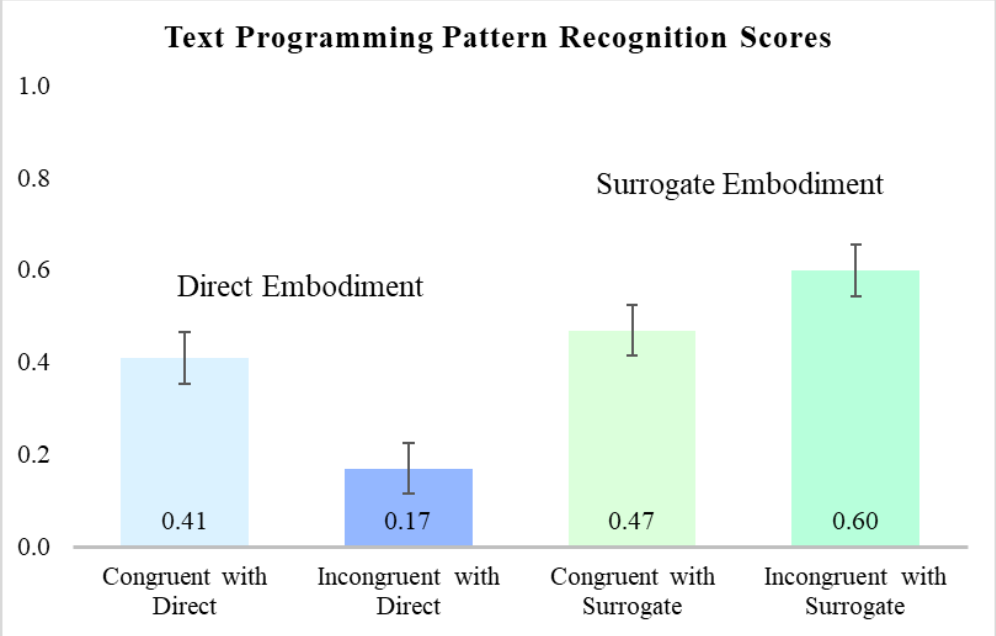


Figure 14. Text-based block programming test pattern recognition scores by group

When looking at total scores across the three concepts in the text programming test, no statistical significance was found. However, the difference between the Direct embodiment group (M = 0.89, SD = 1.430) and Surrogate embodiment group (M = 1.49, SD = 1.652) on text programming was significant, indicating that surrogate body movement significantly affected text programming positively (Table 18).

Table 18

Means and Standard Deviations of Text-based Block Programming Scores by Group

	Direct Embodiment		Surrogate Embodiment		Total	
	M	*SD*	*M*	*SD*	*M*	*SD*
Congruent Gesture	0.88	1.269	1.32	1.635	1.11	1.469
Incongruent Gesture	0.89	1.605	1.65	1.694	1.29	1.675
Total	0.89*	1.430	1.49*	1.652	1.20	1.570

* $p < .05$, *Note*. Maximum score is 5.

Moreover, the correlation between the graphic-based and text-based block programming tests was found to be significant on several items (Table 19). There is a strong positive correlation between matching skills (debugging and debugging (p = .001) in both tests.

Table 19

Pearson Correlations between Graphic Programming and Text Programming

		Graphic Programming			Text Programming		
		Sequence	Pattern Recognition	Debugging	Sequence	Pattern Recognition	Debugging
Graphic Programming	Sequence	_					
	Pattern Recognition	.019*	_				
	Debugging	.000**	.023*	_			
Text Programming	Sequence	.098	.085	.158	_		
	Pattern Recognition	.096	.547	.036*	.414	_	
	Debugging	.000**	.750	.001**	.014*	.003**	_

**Correlation is significant at the 0.01 level (2-tailed).
* Correlation is significant at the 0.05 level (2-tailed).

These results answered the following hypotheses.

H2.2. The Incongruent gesture with Surrogate embodiment group will be more likely perform better on text-based block programming.

Although the results found that participants in the Incongruent gesture with Surrogate embodiment group scored the highest on text-based block programing, no significant difference was found among the four intervention groups.

H3. Learners in the Incongruent gesture group will more likely show a greater degree of text-based block programming.

We could not find any significant differences between the Congruent gesture and Incongruent gesture groups.

H4. The Surrogate embodiment group will demonstrate greater performance across all outcomes compared to the Direct embodiment group, except on persistence.

It was confirmed from the results that those who used surrogate embodiment did significantly better on text programming than those who used direct embodiment.

H4.1. Learners with better skills in graphic-based block programming will show higher achievement on text-based block programming.

We found a significantly positive correlation between graphic- and text-based block programming performance.

Self-Efficacy Survey

The four questions on the self-efficacy survey asked about self-conception related to programming and iPad usage. The pre-test results confirmed that participants in each group did not significantly differ on self-efficacy, F (3, 77) = .242, p = .867.

When looking at learning outcomes from self-efficacy, only Question 3 (i.e., I feel confident that I can figure out how to use new features of an iPad coding app on my own) indicated a significant difference between pre- and post-test scores across the four experimental groups. However, there was no significant learning among four groups or between the two main factors.

A 2x2 ANCOVA discovered that Question 3, which measures how confident one is in using programming applications, displayed a significant interaction effect between the two factors of hand gesture and embodiment, F (1, 67) = 5.886, p = .018, η^2= .081 (Table 20).

Table 20

Two-way ANCOVA on Self-Efficacy Post-Survey Q3 Scores by Gesture and Embodiment Factor

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Gesture	5.152	1	5.152	4.421	0.039*	0.062
Embodiment	1.494	1	1.494	1.282	0.262	0.019
Gesture*Embodiment	6.860	1	6.860	5.886	0.018*	0.081
Error	78.083	67	1.165			

$R^2 = .135$(Adj. $R^2 = .097$), * $p < .05$

Within the Congruent gesture group on Question 3, those who used direct embodiment showed higher self-efficacy than those using surrogate embodiment. On the other hand, the Direct embodiment group on Question 3 indicated lower self-efficacy than the Surrogate embodiment group within incongruent gesture. The Congruent gesture with Direct embodiment group achieved the highest self-efficacy scores among the four groups, followed by the Incongruent with Surrogate group. The Incongruent gesture with Direct embodiment group demonstrated the lowest self-efficacy scores (Figure 15). However, a post-hoc one-way ANOVA did not reveal significant differences between each of the pairwise comparisons.

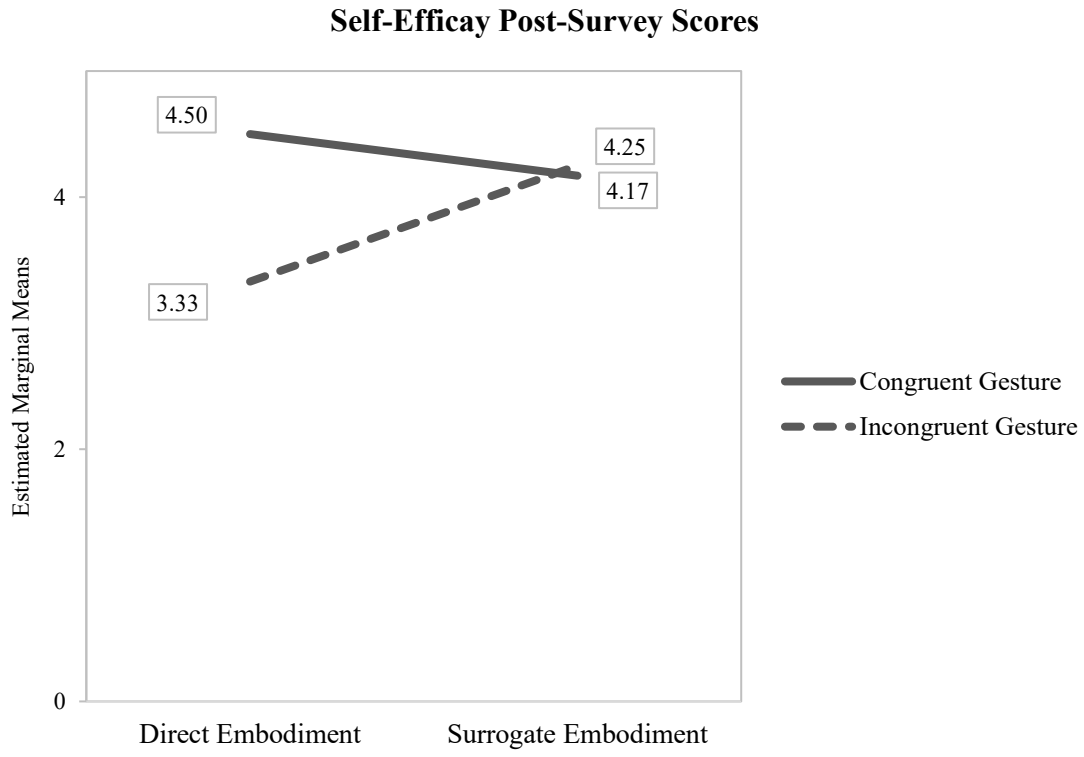


Figure 15. Self-efficacy post-survey Q3 scores by group

Also on Question 3, the gesture factor revealed a significant difference, F (1, 67) = 4.421, p = .039, η^2= .062 (Table 20). Students who practiced with congruent gesture showed higher self-efficacy than those who used incongruent gesture. The estimated difference on Question 3 between congruent gesture and incongruent gesture was 0.47, which was significant (Figure 15). This result is opposite to the graphic-based block programming total score analysis.

In order to evaluate the relationship between self-efficacy and programming abilities, Pearson correlations among the self-efficacy, graphic, and text programming post-tests found that the self-efficacy and text programming test items, specifically self-

efficacy Question 4 (understanding) and text programming sequence skills ($p = .006$), are correlated. However, self-efficacy and the graphic programming test items are not significantly correlated.

These results addressed the following hypotheses.

H1. Participants will have more self-efficacy after debugging activities through gestures and embodiments.

Although self-efficacy enhancement occurred between pre- and post-tests across the four experimental groups, no significant differences were found.

H2.3. Learners who participate in the Congruent gesture and Surrogate embodiment groups will show higher levels of self-efficacy.

In contrast to the hypothesis, the Congruent gesture and Surrogate embodiment groups did not perform the best on self-efficacy.

H.3. Learners in the Congruent gesture group will exhibit higher self-efficacy than the Incongruent group.

The results show that the Congruent group indicated significantly higher self-efficacy than those in the Incongruent group on Question 3. However, the Congruent group did not show significantly higher self-efficacy than the Incongruent group on the other three survey items.

H.4. The Surrogate embodiment group will demonstrate greater performance across all outcomes on the embodiment variables, except persistence, compared to the Direct embodiment group.

We could not find any significance between the Direct and Surrogate embodiment groups.

H5.2. Higher self-efficacy will correlate with higher programming skills.

No significant correlations were found between self-efficacy and graphic programming skills except between self-efficacy and text programming sequence skills, which were correlated.

Persistence Task

A persistence task was conducted to see whether failure experiences in the embodied debugging intervention were productive for future performance in learning, by showing resilience and persistence in the face of failure. After completion of the debugging intervention, students were asked to play games in Kodable, a children's programming application, on an iPad for 20 minutes and freely raise their hands if they wanted to stop playing.

A two-way ANOVA on their persistence duration revealed a significant main effect of embodiment on persistence, F (1, 50) = 4.841, p = .032, η^2= .088, but no significant main effect of gesture and no interaction between gesture and embodiment (Table 21).

Table 21

Two-way ANOVA on Persistence Task Duration by Gesture and Embodiment

Source	SS	df	MS	F	Sig.	Partial Eta Squared
Gesture	24673.776	1	24673.776	0.217	0.643	0.004
Embodiment	549286.664	1	549286.664	4.841	0.032*	0.088
Gesture*Embodiment	11506.070	1	11506.070	0.101	0.751	0.002
Error	5672944.88	50	113458.898			

$R^2 = .102$ (Adj. $R^2 = .048$), * $p < .05$

When comparing estimated marginal means between the embodiment groups, the Direct embodiment group ($M = 975.50$, $SD = 274.455$) showed stronger persistence than the Surrogate embodiment group ($M = 757.54$, $SD = 372.916$), with a mean difference of 217.96 seconds on the persistence task (Figure 16).

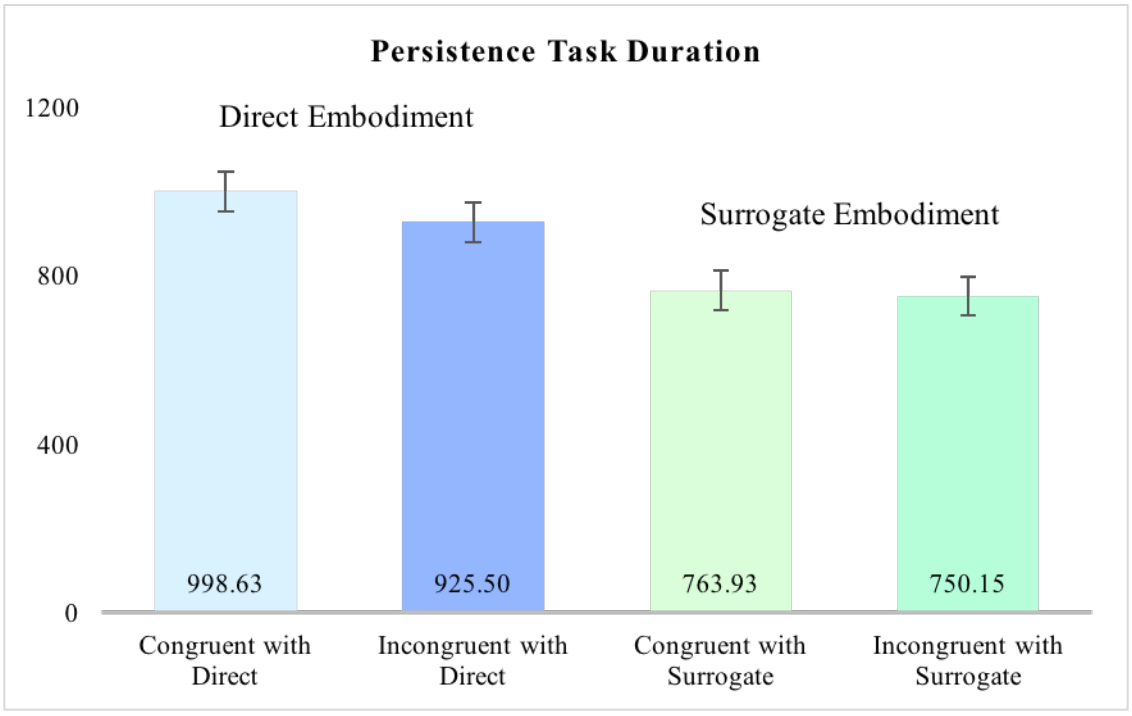


Figure 16. Persistence task duration by group

In regards to the relationship between self-efficacy and persistence, no significant relationships were found between persistence and the two programming tests.

These results addressed the following hypotheses.

H.2.4. Learners in the Incongruent gesture group with Direct embodiment group will show a greater degree of persistence.

There were no significant differences among the four intervention groups on level of persistence.

H.3. Learners in the Incongruent gesture group will more likely show a greater degree of persistence.

The results revealed no significant difference in persistence between the two gesture groups.

H.4. The Surrogate embodiment group will demonstrate greater performance across all outcomes on the embodiment variables, except persistence, compared to the Direct embodiment group.

The results confirmed that students in the Direct embodiment group demonstrated a higher level of persistence.

H5.3. Higher persistence will correlate with higher programming skills.

No significant correlations were found between persistence and programming skills.

Discussion

The purpose of this study was to examine hand gesture and embodiment on the development of young students' computational thinking skills (i.e., graphic and text programming skills) and 2) personal attributes (i.e., self-efficacy, persistence) after an unplugged debugging intervention. The two factors include different types of hand gesture (congruent and incongruent gesture) and different types of embodiment (direct and surrogate embodiment). The study was interested in the individual effects of each factor as well as the combined effects from the two factors.

Here is a brief summary of how each factor was operationalized in the unplugged debugging activities throughout this study. As a baseline, students' prior knowledge on programming (graphic-based block programming) and self-efficacy were measured. Participants then completed an unplugged debugging intervention under one of four intervention groups (2x2 research design, two variables under the gesture condition, two variables under the embodiment condition). For the first phase of debugging, students used one of two types of a laminated debugging worksheet according to their assigned gesture group (Congruent or Incongruent gesture group). They fixed and revised a given debugging worksheet that had several errors. Students in the Congruent gesture group detached and replaced Velcro coding blocks onto the worksheet, which was compatible to the maze path in which a character would move according to the coding script the student made. On the other hand, the Incongruent group debugged with Velcro coding blocks from left to right order, which is incongruent with the maze path.

After revising the code on the worksheet, students tested the revised code through different forms of body movement (direct or surrogate embodiment). The Direct embodiment group directly moved the character on the worksheet maze following the revised code while verbalizing the code, whereas students in the Surrogate embodiment group gave verbal commands from the revised code to the researcher, who moved the character on the worksheet. In order to examine the effect of the unplugged debugging intervention, post-tests on programming, self-efficacy, and persistence were administered.

The following section discusses the results in four categories 1) programming assessments (i.e., graphic- and text-based block programming); 2) personal attributes (i.e., self-efficacy and persistence); 3) learning outcomes between pre- and post-tests; and 4) correlations among the four tests in the study. Interpretations are made by comparing the main effect of the two factors and the four experimental groups to each other.

Programming Assessments: Graphic and Test Programming

The Incongruent group outperformed the Congruent group on graphic programming. A possible explanation why the use of incongruent gesture resulted in better performance than congruent gesture on the graphic programming test can be "desirable difficulties" in the learning environment (Bjork, 2011), in that explicitly incorporated inconsistency or ambiguity in the learning materials promotes greater higher-level thinking than simpler materials (Byrge & Goldstone, 2011; Mannes & Kintsch, 1987; Martin & Schwartz, 2005), as discussed in Chapter II. Another possibility is conditional similarity where there is similarity between incongruent gesture and the

graphic-based block programming test format as most children's programming applications are designed to arrange coding blocks in left to right order. This could have contributed to better performance on the graphic programming test by those who employed incongruent gesture. For future research, measures should include features from congruent gesture to overcome this test limitation.

In regards to text-based block programming, the hypothesis was validated in that surrogate embodiment significantly did better on text programming than direct embodiment. Although both the graphic programming and text programming tests in the study examined the same computational thinking skills of sequence, pattern recognition, and debugging, the data analysis revealed inconsistent outcomes. There was a main effect of gesture on graphic programming, whereas there was a main effect of embodiment on text programming.

A possible explanation is cognitive overload during the text programming test. Students who employed surrogate embodiment had higher scores than those who used direct embodiment. Most participants never experienced text-based block programming before they were given the text programming post-test. Thus, to solve the text programming test, it may have been easier for students to employ CT skills learned from surrogate embodiment, which require less cognitive capacity than direct embodiment. In other words, such CT concepts learned from direct embodiment likely interfered with students fully applying their CT skills to solve text programming. Although there was conditional similarity between direct embodiment and the interface of most children's programming applications on a touch-based tablet in the way children need fingers to

manipulate characters, this did not result in better text programming performance. This emphasizes any cognitive process associated with this result.

In contrast to what was expected, no main effects of hand gesture were found in text-based block programming and no main effects of embodiment were found in graphic-based block programming. For the programming assessments, a graphic programming test was used to examine the domain students practiced through the debugging intervention whereas a text programming test was used to see any transfer of learned content. Results revealed efficiency of surrogate embodiment. Since the graphic and text programming tests in the study assessed the same competencies of sequence, pattern recognition, and debugging, it was not expected that the impact of the two factors would differ between graphic and text programming.

There are two possible explanations for the dissimilar results between the two programming tests. During the embodied debugging intervention, students initially had physical embodiment through congruent or incongruent gesture to fix the code on the worksheet and then test it through direct or surrogate embodiment. Based on a study from Black et. al (2012), the assumption is that after students physically embody to debug the programming code, observing surrogate behavior activities increases their learning, understanding, and motivation. Such surrogate experience results in students thinking about and interacting with the world that will lead to greater transfer of learning beyond the classroom setting. Another possible explanation is the different interfaces of graphic programming and text programming. In most text-based block programming applications, the coding block is arranged in a top-down order, not left to right order of graphic

programming applications. Thus, the association between perception and action is possibly different between graphic and text programming.

Personal Attributes: Self-efficacy and Persistence

Data analysis of the self-efficacy survey suggests that the combination of hand gesture and embodiment showed significant results. On self-efficacy within the Congruent gesture group, direct embodiment showed higher self-efficacy than those in the Surrogate embodiment group. On the other hand, the Direct embodiment group had lower self-efficacy than the Surrogate embodiment group when using incongruent gesture (Figure 15). It seems likely that congruent gesture effectively worked with direct embodiment, but not as effectively as with surrogate embodiment, on promoting self-efficacy. Incongruent gesture complemented surrogate embodiment in self-efficacy development, but was less promising when combined with direct embodiment. In short, congruent gesture with direct embodiment was demonstrated to be effective on self-efficacy. This may be attributed to the fact that situated actions from direct embodiment, accompanied with more explicit congruent gestures, are demonstrated to be the preferred self-efficacy strategy. Having authentic experience through one's own body movement with direct embodiment can increase perceived accomplishment and competence. Such perceptions that are directly linked to self-efficacy (Bandura, 1986) promote more positive self-efficacy.

Also, comparison of the two gestures to each dependent variable revealed that the Incongruent group outperformed the Congruent group on graphic programming. The Congruent group scored higher on one of the self-efficacy survey items (Q5. Confidence

in using programming application) than the Incongruent group. These results are consistent with the hypothesis.

Regarding the persistence task, the Direct embodiment group showed a higher level of persistence than those in the Surrogate embodiment group. This result was consistent with the hypothesis. As reviewed earlier, more cognitive load associated with direct embodiment possibly yields higher persistence in the face of difficulty.

Learning Outcomes after Debugging Intervention

The first research question assessed whether any learning occurred during the embodied debugging intervention in which different hand gestures and embodiment were utilized. The learning outcome measured programming skills and personal development by comparing pre- and post-test scores of the graphic programming test and self-efficacy survey. The results indicated that learning occurred after the debugging intervention from graphic programming and self-efficacy Question 3 (i.e., I feel confident that I can figure out how to use new features of an iPad coding app on my own) across the four experimental groups. However, no significant difference in learning was found among the four groups or from the two main factors.

The hypothesis was partially rejected on graphic programming and self-efficacy. Although statistical significance was found when comparing the pre- and post-tests of all participants, we cannot tell if the two factors of hand gesture and embodiment have different effects on graphic programming learning.

A possible explanation is that the combination of gesture and embodiment showed contrasting results, as was seen in the results of self-efficacy. Congruent gesture

did not work favorably with direct embodiment but complemented surrogate embodiment. Instead, incongruent gesture worked effectively with direct embodiment and was less promising when combined with surrogate embodiment.

The Relationship among the Four Tests

The fifth research question examined the relationship among the dependent variables. In particular, this study was interested in the relationships between: 1) graphic-based block programing and text-based block programming; 2) self-efficacy on the two programing tests; and 3) persistence on the two programming tests.

This study found a significantly positive correlation between graphic- and text-based block programming performance. From this result, it was concluded that cognitive skills in the domain of CT (sequencing, problem solving, and predicting) from graphic-based programming in early grades can be a possible predictor for programming skills in the future. These results confirm previous research which agrees that teaching computer programming can be a way to train CT (Grover & Pea, 2013; Kafai & Burke, 2013; Lye & Koh, 2014; ; Mannila, Dagiene, Demo, Grgurina, Mirolo, Rolandsson, & Settle, 2014) and to apply CT skills (Orr, 2009, August). This result sheds light on the importance of integrating programming into K-12 education to nurture good computational thinkers.

There was a significantly positive correlation among self-efficacy with text programming performance. One possible explanation for this finding has to do with the interactions between self-efficacy and problem solving. Previous research asserts that higher self-efficacy is critical in problem solving due to its value as risk taking or persistence, which, consequently, leads to higher achievement in the programming

domain (Amabile, 1996; Bandura, 1986; Cross, 2006). However, no significant correlations were found between self-efficacy and the graphic block programming test. Given the fact that text-based programming test was implemented to measure knowledge transfer whereas the graphic-based programming test examined knowledge practiced during the debugging intervention, transfer possibly requires considerable problem solving skills.

VI – CONCLUSION

Throughout the three debugging studies, the researcher was interested in designing instructional practices that cultivate computational thinking (programming and debugging skills, problem solving strategies) and to develop persistence and self-efficacy in the face of problems (i.e., programming errors) through debugging activities. The studies implemented various types of instructional embodiment on unplugged debugging interventions. During the debugging interventions, participants were asked to debug given coding scripts which contained errors they needed to fix.

Study 1 compared degrees of embodiment (full embodiment and low embodiment) with a control group during an unplugged debugging intervention. Study 2 examined the effects of different degrees of embodied activities (full embodiment vs. low embodiment) combined with different types of text-based programming language (coding language vs. narratives) on young students' computational thinking and self-efficacy. Based on the findings from the second study, the unplugged debugging intervention in Study 3 was designed with different gestures (congruent gesture vs. incongruent gesture) accompanied with different types of embodiment (direct embodiment vs. surrogate embodiment).

Study 3 examined action and perception to teach CT skills, cognitive skills related to the programming domain, and to enhance personal development (i.e., self-efficacy, persistence) through congruent or incongruent hand gestures incorporated with different types of embodiment. The types of embodiment were direct embodiment and surrogated embodiment. Direct embodiment refers to learners' whole body movement and surrogate

embodiment is the control of a surrogate through verbal commands (e.g., designed to imitate the human body and other objects in programming education).

There are a number of implications in the education sector that emerge from the findings of these three studies. This section highlights suggestions on instructional design.

Implications for Computational Thinking and Programming Education

Research on children's programming education reviewed earlier argues for the need to simplify, eliminate, or delegate the challenges of programming. One of the instructional methods applied to the three debugging studies were unplugged activities through embodiment. Another was problem solving practices through debugging, as the computational practice of testing and debugging is an essential problem-solving skill in programming (McCauley et al., 2008). Unplugged activities and problem solving are fundamental concepts for higher level computational thinking skills, according to the Computational Thinking Pedagogical Framework (CTPF) (Kotsopoulos, Floyd, Khan, Namukasa, Somanath, Weber, & Yiu, 2017).

The CTPF includes four pedagogical experiences: (1) unplugged, (2) tinkering, (3) making, and (4) remixing. Embodiment in this paper used the unplugged form, defined as "unplugged experiences focus on activities implemented without the use of computers" in the CTPF (Kotsopoulos et al., 2017). Debugging has a commonality with tinkering since "tinkering experiences primarily involve activities that take things apart and engaging in changes and/or modifications to existing objects" (Kotsopoulos et al.,

2017). In order for students to fully engage in CT, all four experiences are necessary. Particularly for novices and depending on the concepts under exploration, a sequential approach among these experiences may be helpful.

Therefore, the instructional approach in this paper can be used as introductory programming lessons for young students or novice learners. That said, it is recommended to introduce computational thinking concepts through various types of embodiment. Also, when considering the sequential approach from the CTPF, providing an opportunity to solve a debugging task prior to noble computational thinking concepts will encourage children to develop computational thinking and problem solving skills.

Another instructional approach from Study 2 was to lower the abstract and complex nature of programming language (e.g., syntax code, graphical code) (Ko & Myers, 2004) through narratives that resemble everyday language. By recognizing potential difficulties of programming applications for young students, instructions need to consider that not all students learn in exactly the same way. To support learners with different learning styles and preferences to experience meaningful learning, the debugging activities in the second study adapted different types of text-based programming language. To create an environment of full participation into computational thinking skills, instructional technology design should provide practice in multiple ways to optimize young students' learning experience. Learning experiences in varied forms will allow accessibility by diverse populations of learners into computational thinking skills.

Instructional Design for Various Disciplines

The widespread adoption of gesture-based computing, such as smartphones and tablets, has rapidly increased the way people use media. One promise of gesture-based computing is touch gestures coherently follow the actions of the user to effectively remember things. In Study 3, participants utilized different gestures (congruent gesture vs. incongruent gesture) to debug given coding scripts on paper which contained errors they needed to fix. Results revealed that congruent gesture was effective for having self-confidence on using programming tools among children. On the other hand, incongruent gesture showed favorable impact on graphic coding blocks.

The use of action can greatly benefit conceptual understanding in STEM disciplines. Since hand movements are so natural and pervasive and have a special connection to the concepts they accompany, researchers that claim gestures are effective in several disciplines to help students instill a profound understanding of abstract concepts, such as language and mathematics (Kelly, Manning, & Rodak, 2008).

Also, effectiveness of gestures toward CT skills from debugging studies opens up potential to various disciplines other than STEM subjects. CT involves concepts (e.g., debugging) primarily from computer science, which are shared across other disciplines, such as science, mathematics, social science, biology, language arts, and engineering (Kafai & Burke, 2013; Lye & Koh, 2014). Even though the finding on different gestures in Study 3 was derived from computer science, CT practices using gesture can be extended to other disciplines. In this sense, incorporating congruent and incongruent

gestures into learning activities can be useful for children when they are introduced to concepts similar to CT in diverse disciplines.

Implications for Emotional Development

The three studies examined the impact of embodiment and programming language type during a debugging intervention to support young students' self-efficacy development along with computational thinking. Results demonstrated that unplugged debugging practices with different types of text representations that contain coding language or narratives lower their cognitive capacity while working on debugging problems compared to working on a touch-based tablet. Curricular design around such an unplugged environment should be rooted in developmentally appropriate teaching practices sensitive to children's social, emotional, physical, and cognitive development (Copple & Bredekamp, 2009). In this sense, leveraging deficits of self-efficacy in programming through debugging intervention encouraged young students new to programming towards higher self-efficacy. In related research, student computer programming self-efficacy positively predicted performance (Ramalingam, LaBelle, & Wiedenbeck, 2004). Similarly, Moos and Azevedo (2009) discovered that students' computer self-efficacy is related to their learning performance in computer-based learning environments. Furthermore, it is expected that positive development of self-efficacy enhances students' academic interest and effort, as opportunities to foster self-efficacy positively correlate with students' academic interest and effort and overall achievement (Pekrun, Goetz, Perry, Kramer, Hochstadt, & Molfenter, 2004). In this

sense, the debugging intervention guided young students new to programming to more positive self-efficacy and persistence. Thus, to develop a high level of self-efficacy in young students, unplugged activities, narratives that are similar to natural everyday language, and congruent gestures are important instructional methods.

Limitations

Although these debugging studies provide instructional approaches to support children's computational thinking, cognitive skills in the domain of programming, and self-efficacy, it is difficult to examine the acquisition of such cognitive skills using some of the tests developed by the researcher. In particular, the main limitation of this study is not employing a validated measurement instrument for computational thinking. As other research in computational thinking has pointed out, there is still a large number of tests related to CT that have yet to undergo a comprehensive psychometric validation process (Mühling, 2015) In the future study, more reliability and validity evidence needs to be collected and reported to help develop assessments for computational thinking.

These studies offer both strengths and weaknesses of integrating programming education into an after-school classroom by incorporating embodied instruction and a different type of text-based programming language. Specific limitations of the three debugging studies are addressed in the discussion section of each study. For future study, learning environments should be created to allow students to actively construct knowledge with deeper understanding through exploration and interaction so that new content is introduced by 1) building on skills and knowledge previously learned; 2)

generating tasks that require metacognitive strategy and reflective thinking; 3) analyzing problems step-by-step to stimulate computational thinking; 4) providing multiple methods of engagement (e.g., body, imagination) to create associative structures between learning and various cues for better information retrieval; 5) maintaining an appropriate level of stimulation with multimedia materials so students require only a modest level of cognitive load; 6) situating classroom activities in familiar and meaningful contexts; 7) providing multiple forms of instructional media representation and encouraging students' expression in multiple ways to support differentiated capacities, including minority students; and 8) fostering the development of positive perception about self, world, and moral character.

REFERENCES

Abrahamson, D., & Lindgren, R. (2014). Embodiment and embodied design. *The Cambridge Handbook of the Learning Sciences, 2,* 358-376.

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin, 37*(3), 84-88.

Alibali, M. W., Spencer, R. C., Knox, L., & Kita, S. (2011). Spontaneous gestures influence strategy choices in problem solving. *Psychological Science, 22*(9), 1138-1144.

Amabile, T. M. (2018). *Creativity in context: Update to the social psychology of creativity*. Routledge.

Ananiadou, K., & Claro, M. (2009). *21st century skills and competences for new millennium learners in OECD countries* (EDU working paper no. 41). OECD Publishing.

Anewalt, K. (2008). Making CS0 fun: An active learning approach using toys, games and Alice. *Journal of Computing Sciences in Colleges, 23*(3), 98-105.

Bandura, A. (1977). Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review, 84*(2), 191.

Bandura, A. (1986). S*ocial foundations of thought and action*. Prentice Hall.

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *Acm Inroads, 2*(1), 48-54.

Barsalou, L. W. (2008). Grounded cognition. *Annual Review of Psychology, 59*, 617-645.

Basawapatna, A. R., Koh, K. H., & Repenning, A. (2010, June). Using scalable game design to teach computer science from middle school to graduate school. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (pp. 224-228).

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology, 13*(1), 20-29.

Bers, M. U., Doyle-Lynch, A., & Chau, C. (2012). Positive technological development: The multifaceted nature of youth technology use towards improving self and society. *Constructing the Self in a Digital World*, 110-136.

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers and Education*, *72*, 145-157.

Biesta, G. J., & William, N. C. B. E. (2003). *Pragmatism and educational research.* Lanham, MD: Rowman and Littlefield.

Binkley, M., Erstad, O., Herman, J., Raizen, S., Ripley, M., Miller-Ricci, M., & Rumble, M. (2012). Defining twenty-first century skills. In P. Griffin, B. McGaw & E. Care (Eds.), *Assessment and teaching of 21st century skills*. Springer.

Bjork, E. L., & Bjork, R. A. (2011). Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society, 2*, 59-68.

Bjork, R. A. (1994). Memory and metamemory considerations in the training of human beings. *Metacognition: Knowing about knowing, 185*.

Bjork, R. A., & Linn, M. C. (2006). The science of learning and the learning of science. *Association of Psychological Science Observer, 19*(3).

Black, J. B. (2007). Imaginary worlds. In M.A. Gluck, J.R. Anderson & S.M. Kosslyn (Eds.), *Memory and mind*. Mahwah, NJ: Lawrence Erlbaum Associates.

Black, J. B. (2010). An embodied/grounded cognition perspective on educational technology. In M. S. Khine & I. M. Saleh (Eds.), *New science of learning* (pp. 45-52). Springer.

Black, J. B., Segal, A., Vitale, J., & Fadjo, C. (2012). Embodied cognition and learning environment design. *Theoretical Foundations of Learning Environments*, 198-223.

Bowers, L., Huisingh, R., & LoGiudice, C. (2005). *TOPS 3, Elementary: A test of reasoning in context skill area problem solving and reasoning, developmental ages 6-0 through 12-11 years*. LinguiSystem.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). *How people learn*. Washington, DC: National Academy Press.

Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking.* Paper presented at the Proceedings of the AERA Annual Conference, Vancouver, Canada.

Broaders, S. C., Cook, S. W., Mitchell, Z., & Goldin-Meadow, S. (2007). Making children gesture brings out implicit knowledge and leads to learning. *Journal of Experimental Psychology: General, 136*(4), 539.

Byrge, L., & Goldstone, R. (2011). *Distinguishing levels of grounding that underlie transfer of learning.* Paper presented at the Proceedings of the Annual Meeting of the Cognitive Science Society.

Carver, S. M., & Klahr, D. (1986). Assessing children's LOGO debugging skills with a formal model. *Journal of Educational Computing Research, 2*(4), 487-525.

Casasanto, D., & Dijkstra, K. (2010). Motor action and emotional memory. *Cognition, 115*(1), 179-185.

Chandler, P., & Sweller, J. (1996). Cognitive load while learning to use a computer program. *Applied Cognitive Psychology, 10*(2), 151-170.

Chu, M., & Kita, S. (2008). Spontaneous gestures during mental rotation tasks: Insights into the microdevelopment of the motor strategy. *Journal of Experimental Psychology: General, 137*(4), 706.

Clark, D., & Linn, M. C. (2003). Designing for knowledge integration: The impact of instructional time. *The Journal of the Learning Sciences, 12*(4), 451-493.

Cooper, S., Dann, W., & Pausch, R. (2000). *Alice: A 3-D tool for introductory programming concepts.* Paper presented at the Journal of Computing Sciences in Colleges.

Copple, C., & Bredekamp, S. (2009). *Developmentally appropriate practice in early childhood programs serving children from birth through age 8*. Washington, DC: National Association for the Education of Young Children.

Cross, N. (2006). *Designerly Ways of Knowing.* Springer.

Fadjo, C. L. (2012). *Developing computational thinking through grounded embodied cognition* [Doctoral dissertaton, Columbia University]. ProQuest Dissertations Publishing.

Fadjo, C. L., Hong, J., Chang, C., Geist, E., & Black, J. B. (2010). *An embodied approach to the instruction of conditional logic in video game programming.* Paper presented at the Ed-Media: World Conference on Educational Multimedia, Hypermedia & Telecommunications, Toronto, Canada.

Fadjo, C. L., Lu, M., & Black, J. B. (2009). *Instructional embodiment and video game programming in an after school program.* Paper presented at the World Conference on Educational Multimedia, Hypermedia and Telecommunications, Chesapeake, VA.

Fadjo, C., Shin, J., Lu, M., Chan, M., & Black, J. B. (2008). *Embodied cognition and video game programming.* Paper presented at the Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications.

Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). *Designing ScratchJr: Support for early childhood learning through computer programming.* Paper presented at the Proceedings of the 12th International Conference on Interaction Design and Children.

Flavell, J. H. (1979). Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American Psychologist, 34*(10), 906.

Glenberg, A. M. (2010). Embodiment as a unifying perspective for psychology. *Wiley Interdisciplinary Reviews: Cognitive Science, 1*(4), 586-596.

Glenberg, A. M., Gutierrez, T., Levin, J. R., Japuntich, S., & Kaschak, M. P. (2004). Activity and imagined activity can enhance young children's reading comprehension. *Journal of Educational Psychology, 96*(3), 424.

Glenberg, A. M., & Kaschak, M. P. (2002). Grounding language in action. *Psychonomic Bulletin and Review, 9*(3), 558-565.

Goldin-Meadow, S., & Beilock, S. L. (2010). Action's influence on thought: The case of gesture. *Perspectives on Psychological Science, 5*(6), 664-674.

Goldin-Meadow, S., Cook, S. W., & Mitchell, Z. A. (2009). Gesturing gives children new ideas about math. *Psychological Science*, *20*(3), 267-272.

Goldstone, R., Landy, D., & Brunel, L. C. (2011). Improving perception to make distant connections closer. *Frontiers in Psychology, 2*, 385.

Goulet, D. V., & Slater, D. (2009). Alice and the introductory programming course: An invitation to dialogue. *Information Systems Journal, 7*(20), 3-16.

Grover, S., & Pea, R. (2013). Computational thinking in K–12 A review of the state of the field. *Educational Researcher, 42*(1), 38-43.

Guzdial, M. (2014, October 15). Teaching computer science better to get better results. *Computing Education Research Blog*. https://computinged.wordpress.com/2014/10/15/we-need-to-fix-the-computer-science-teaching-problem/

Hayes-Roth, B., & Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science, 3*(4), 275-310.

Hegarty, M. (2011). The cognitive science of visual-spatial displays: Implications for design. *Topics in Cognitive Science, 3*(3), 446-474.

Holbert, N. R., & Wilensky, U. (2011). Formula tracing: Designing a game for kinematic exploration and computational thinking. In Steinkuehler, K., Martin, C., Ochsner, A. (Eds.), *Proceedings of the 7th Annual Games , Learning, and Society Conference.* Madison, WI.

Horn, M. S., Crouser, R. J., & Bers, M. U. (2012). Tangible interaction and learning: The case for a hybrid approach. *Personal and Ubiquitous Computing, 16*(4), 379-389.

Hostetter, A. B., & Alibali, M. W. (2008). Visible embodiment: Gestures as simulated action. *Psychonomic Bulletin and Review, 15*(3), 495-514.

Ioannidou, A., Bennett, V., Repenning, A., Koh, K. H., & Basawapatna, A. (2011). *Computational thinking patterns*. Paper presented at the AERA Annual meeting, New Orleans, Louisiana.

Jamalian, A., Giardino, V., & Tversky, B. (2013). *Gestures for thinking.* Paper presented at the Proceedings of the Annual Meeting of the Cognitive Science Society.

Johnson-Glenberg, M. C., Birchfield, D. A., Tolentino, L., & Koziupa, T. (2014). Collaborative embodied learning in mixed reality motion-capture environments: Two science studies. *Journal of Educational Psychology, 106*(1), 86.

Kafai, Y. B., & Burke, Q. (2013). Computer programming goes back to school. *Phi Delta Kappan, 95*(1), 61-65.

Katai, Z., & Toth, L. (2010). Technologically and artistically enhanced multi-sensory computer-programming education. *Teaching and TeacherEeducation, 26*(2), 244-251.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR), 37*(2), 83-137.

Kelleher, C., Pausch, R., & Kiesler, S. (2007). *Storytelling alice motivates middle school girls to learn computer programming.* Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.

Kelly, S. D., Manning, S. M., & Rodak, S. (2008). Gesture gives a hand to language and learning: Perspectives from cognitive neuroscience, developmental psychology and education. *Language and Linguistics Compass, 2*(4), 569-588.

Kirsh, D. (1995). The intelligent use of space. *Artificial Intelligence, 73*(1-2), 31-68.

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*(3), 362-404.

Ko, A. J. (2009). *Attitudes and self-efficacy in young adults' computing autobiographies.* Paper presented at the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing.

Ko, A. J., & Myers, B. A. (2004). *Designing the whyline: A debugging interface for asking questions about program behavior.* Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.

Kotsopoulos, D., Floyd, L., Khan, S., Namukasa, I. K., Somanath, S., Weber, J., & Yiu, C. (2017). A pedagogical framework for computational thinking. *Digital Experiences in Mathematics Education, 3*(2), 154-171.

Lambert, L., & Guiffre, H. (2009). Computer science outreach in an elementary school. *Journal of Computing Sciences in Colleges, 24*(3), 118-124.

Law, L.-C. (1998). A situated cognition view about the effects of planning and authorship on computer program debugging. *Behaviour & Information Technology, 17*(6), 325-337.

Lee, M. J. (2015). *Teaching and engaging with debugging puzzles.* [Doctoral dissertation, University of Washington]. https://digital.lib.washington.edu/researchworks/handle/1773/33985

Lehrer, R., Lee, M., & Jeong, A. (1999). Reflective teaching of logo. *The Journal of the Learning Sciences, 8*(2), 245-289.

Lindgren, R., & Johnson-Glenberg, M. (2013). Emboldened by embodiment six precepts for research on embodied learning and mixed reality. *Educational Researcher, 42*(8), 445-452.

Lu, J. J., & Fletcher, G. H. (2009). Thinking about computational thinking. *ACM SIGCSE Bulletin, 41*(1), 260-264.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior, 41*, 51-61.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008, March). Programming by choice: Urban youth learning programming with scratch. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 367-371).

Manches, A., & Plowman, L. (2015). Computing education in children's early years: A call for debate. *British Journal of Educational Technology, 48*(1), 191-201.

Mannes, S. M., & Kintsch, W. (1987). Knowledge organization and text organization. *Cognition and Instruction, 4*(2), 91-115.

Mannila, L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). *Computational thinking in K-9 education.* Paper presented at the Proceedings of the Working Group Reports of the Innovation & Technology in Computer Science Education Conference.

Margolis, J., Goode, J., & Bernier, D. (2011). The need for computer science. *Educational Leadership, 68*(5), 68-72.

Martin, T., & Schwartz, D. L. (2005). Physically distributed learning: Adapting and reinterpreting physical environments in the development of fraction concepts. *Cognitive Science, 29*(4), 587-625.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education, 18*(2), 67-92.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with scratch. *Computer Science Education, 23*(3), 239-264.

Moos, D. C., & Azevedo, R. (2009). Learning with computer-based learning environments: A literature review of computer self-efficacy. *Review of Educational Research, 79*(2), 576-600.

Moreno, J. (2012). Digital competition game to improve programming skills. *Journal of Educational Technology & Society, 15*(3), 288-297.

Mühling, A., Ruf, A., & Hubwieser, P. (2015, November). *Design and first results of a psychometric test for measuring basic programming abilities*. Proceedings of the Workshop in Primary and Secondary Computing Education (pp. 2-10).

National Research Council. (2010). *Report of a workshop on the scope and nature of computational thinking*. National Academies Press.

Nusen, N., & Sipitakiat, A. (2011). *Robo-blocks: A tangible programming system with debugging for children.* Paper presented at the Proceedings of the 19th International Conference on Computers in Education. Chiang Mai.

Orr, G. (2009). *Computational thinking through programming and algorithmic art.* Paper presented at the SIGGRAPH.

Overmars, M. (2004). Teaching computer science through game design. *Computer, 37*(4), 81-83.

Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. *End-User Development*, 31-50.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Pea, R. D. (1983). Logo programming and problem solving [Technical Report No. 12], *American Educational Research Association Symposium*. Montreal, Canada.

Pekrun, R., Goetz, T., Perry, R. P., Kramer, K., Hochstadt, M., & Molfenter, S. (2004). Beyond test anxiety: Development and validation of the Test Emotions Questionnaire (TEQ). *Anxiety, Stress & Coping, 17*(3), 287-316.

Phillips, P. (2009). Computational Thinking: A problem-solving tool for every classroom. *Communications of the CSTA, 3*(6), 12-16.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). *Self-efficacy and mental models in learning to program.* Paper presented at the ACM SIGCSE Bulletin.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research, 19*(4), 367-381.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., & Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137-172.

Scaffidi, C., Shaw, M., & Myers, B. (2005). *Estimating the numbers of end users and end user programmers.* Paper presented at the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing.

Schwartz, D. L., & Black, J. B. (1996). Shuttling between depictive models and abstract rules: Induction and fallback. *Cognitive Science, 20*(4), 457-497.

Schwartz, D. L., & Black, T. (1999). Inferences through imagined actions: Knowing by simulated doing. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 25*(1), 116.

Segal, A. (2011). *Do gestural interfaces promote thinking? Embodied interaction: Congruent gestures and direct touch promote performance in math* [Doctoral dissertaton, Columbia University]. ProQuest Dissertations Publishing.

Segal, A., Tversky, B., & Black, J. (2014). Conceptually congruent actions can promote thought. *Journal of Applied Research in Memory and Cognition, 3*(3), 124-130.

Shavelson, R. J., & Towne, L. (2002). *Scientific research in education*. National Academies Press.

Sipitakiat, A., & Nusen, N. (2012). *Robo-Blocks: Designing debugging abilities in a tangible programming system for early primary school children.* Paper presented at the Proceedings of the 11th International Conference on Interaction Design and Children.

Smith, D. C., Cypher, A., & Tesler, L. (2000). Programming by example: Novice programming comes of age. *Communications of the ACM, 43*(3), 75-81.

Søndergaard, H., & Mulder, R. A. (2012). Collaborative learning through formative peer review: Pedagogy, programs and potential. *Computer Science Education, 22*(4), 343-367.

Sternberg, R., & Sternberg, K. (2016). *Cognitive psychology*. Nelson Education.

Subrahmaniyan, N., Kissinger, C., Rector, K., Inman, D., Kaplan, J., Beckwith, L., & Burnett, M. (2007). *Explaining debugging strategies to end-user programmers.* Paper presented at the IEEE Symposium on Visual Languages and Human-Centric Computing.

Sung, W., Ahn, J., & Black, J. B. (2017). Introducing computational thinking to young learners: Practicing computational perspectives through embodiment in mathematics education. *Technology, Knowledge and Learning, 22*(3), 443-463.

Sung, W., Ahn, J., Kai, S. M., Choi, A., & Black, J. B. (2016). Incorporating touch-based tablets into classroom activities: Fostering children's computational thinking through iPad integrated instruction. In Mentor D. (Ed.), *Handbook of Research on Mobile Learning in Contemporary Classrooms* (pp. 378-406). IGI Global.

Tew, A. E., Dorn, B., Leahy Jr, W. D., & Guzdial, M. (2008). Context as support for learning computer organization. *Journal on Educational Resources in Computing, 8*(3), 8.

Tversky, B. (2011). Visualizing thought. *Topics in Cognitive Science, 3* (3), 499-535.

Vitale, J. (2012). *Promoting the development of an integrated numerical representation through the coordination of physical materials* [Doctoral dissertaton, Columbia University]. ProQuest Dissertations Publishing.

Wang, D., Qi, Y., Zhang, Y., & Wang, T. (2013). *TanPro-kit: A tangible programming tool for children.* Paper presented at the Proceedings of the 12th International Conference on Interaction Design and Children.

Webb, N. M., Ender, P., & Lewis, S. (1986). Problem-solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal, 23*(2), 243-261.

Wickelgren, W. A. (1974). *How to solve problems: Elements of a theory of problems and problem solving*. San Francisco: W.H. Freeman.

Wilson, A., & Moffat, D. C. (2010). *Evaluating Scratch to introduce younger school children to programming.* Paper presented at the PPIG.

Wilson, M. (2002). Six views of embodied cognition. *Psychonomic Bulletin and Review, 9*(4), 625-636.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717-3725.

Wohl, B., Porter, B., & Clinch, S. (2015). *Teaching computer science to 5-7 year-olds: An initial study with Scratch, Cubelets and unplugged computing.* Paper presented at the Proceedings of the Workshop in Primary and Secondary Computing Education.

Wolman, J., Campeau, P., Dubois, P., Mithaug, D., & Stolarski, V. (1994). *AIR self-determination scale and user guide*. Palo Alto, CA: American Institute for Research.

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry, 17*(2), 89-100.

Wyeth, P. (2008). How young children learn to program with sensor, action, and logic blocks. *The Journal of the Learning Sciences, 17*(4), 517-550.

Yadav, A., Zhou, N., Mayfield, C., Hambrusch, S., & Korb, J. T. (2011). *Introducing computational thinking in education courses.* Paper presented at the Proceedings of the 42nd ACM Technical Symposium on Computer Science Education.

Yang, Y.-F. (2010). Students' reflection on online self-correction and peer review to improve writing. *Computers and Education, 55*(3), 1202-1210.

Zhao, J. (2018). *Using gestures and body movements for thinking and learning* [Doctoral dissertaton, Columbia University]. ProQuest Dissertations Publishing.

Appendix A

Study One: Self-Efficacy Survey Items

• Each question has 5 possible responses
• Choose the response that is most true of you and your life.

- I feel good about myself when using the computer.

1	2	3	4	5
				

- I can express my ideas, my thought, and myself by using the computer.

1	2	3	4	5
				

- If my plan doesn't work, I try another one to meet my goals.

1	2	3	4	5
				

- I enjoy working with friends.

1	2	3	4	5
				

- I am more comfortable working alone.

1	2	3	4	5
				

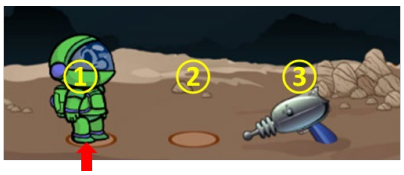
- Maybe, I can be a programmer.

1	2	3	4	5
				

Appendix B

Study Two: Debugging Test Items

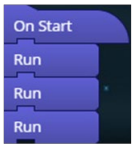
1. If an astronaut follow codes on the left, where would he be arrived?
Choose one of the numbers on the picture.

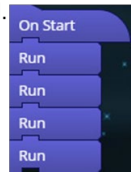


2. How would you change the codes for an astronaut get the ray gun?
Choose one of the following.

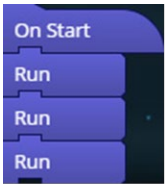
a. 

b. 

c. 

d. 

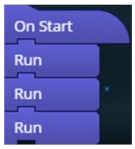
3. If follow the code, where would an astronaut be arrived?
Choose one of the numbers on the picture.

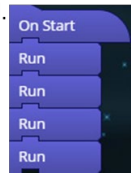


4. How would you change the code to make an astronaut get the ray gun?
Choose one of the following.

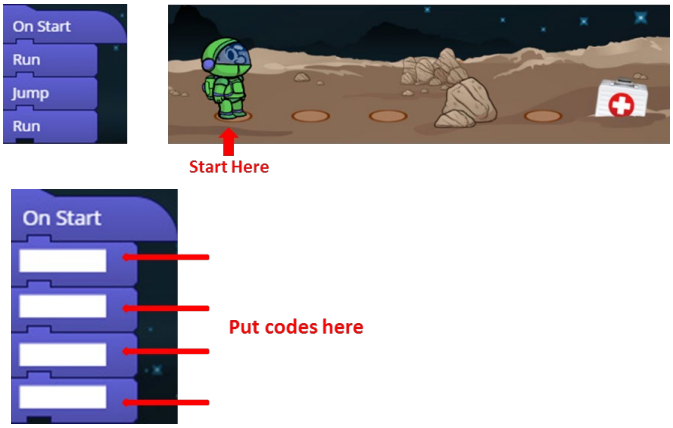
a. 

b. 

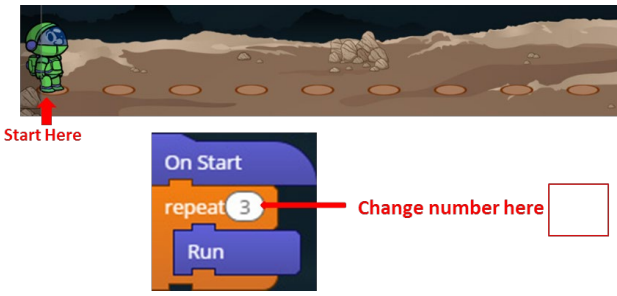
c. 

d. 

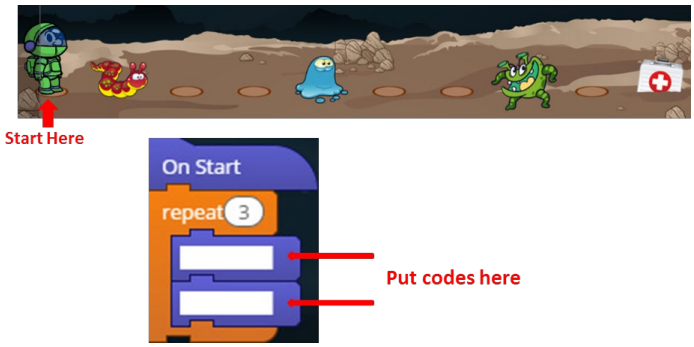
5. **Change CODES** to get the medkit, but avoid rocks using “Jump”.



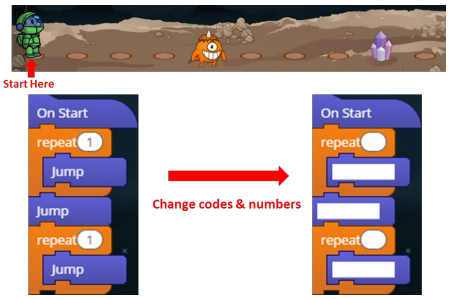
6. **Change NUMBER** in the blank to get the power cell to power the spaceship.



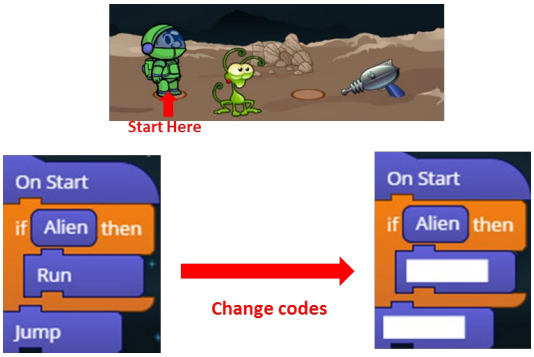
7. Put CODES in the blanks to get the medkit.
Jump if there is an alien.



8. **CHANGE NUMBERS and Codes** in the blanks to get the crystal.
Jump if there is an alien.



9. **Change CODES** to get the raygun. Jump if there is an alien.



10. **Change CODES** to get the raygun. Jump if there is an alien.

Appendix C

Study Two: Self-Efficacy Survey Items

Please think about <u>your CURRENT ability to use a computer</u>, and choose how well you can do each of the following <u>NOW</u>.

- I can easily learn how to use a new computer application

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- When using computer application, I can easily identify the information I need to complete a task.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- When I get stuck, I can easily find information I need from a user manual or the Web.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

Please think about <u>your CURRENT ability to code</u>, and choose how well you can do each of the following <u>NOW</u>.

- I am comfortable with writing code to solve a problem.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I can express my ideas and myself by writing code.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I am able to make a coding project from an idea to a finished work.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I can fix the code when something goes wrong.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- It is important for me to teach other kids the code that I already know.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- When working with other kids on coding, I make sure that they understand everything I am doing.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

Please read the following and think about your behavior in classroom. Choose one that best describes you.

- I am more comfortable working alone when working on a programming project.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I am more comfortable working alone in the classroom.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I enjoy do things with other kids.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- In the future, I want to work in science, technology, mathematics, or engineering field.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I feel good about myself when using the computer (iPads).

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

- I like school, and I am doing well.

Not At All	Below Average	Average	Above Average	Extremely
1	2	3	4	5

Appendix D

Study Three: Graphic-based Block Programming Test Items

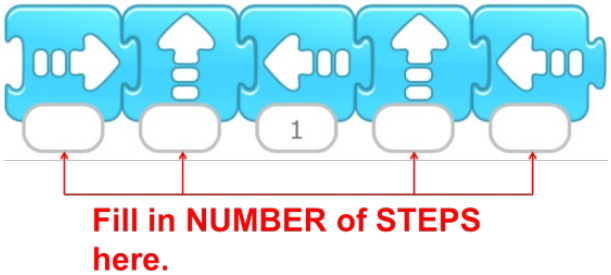
1. A turtle has to catch **Two Jewels**, then go to finish line.
Choose the RIGHT CODE.

a. 

b. 

c. 

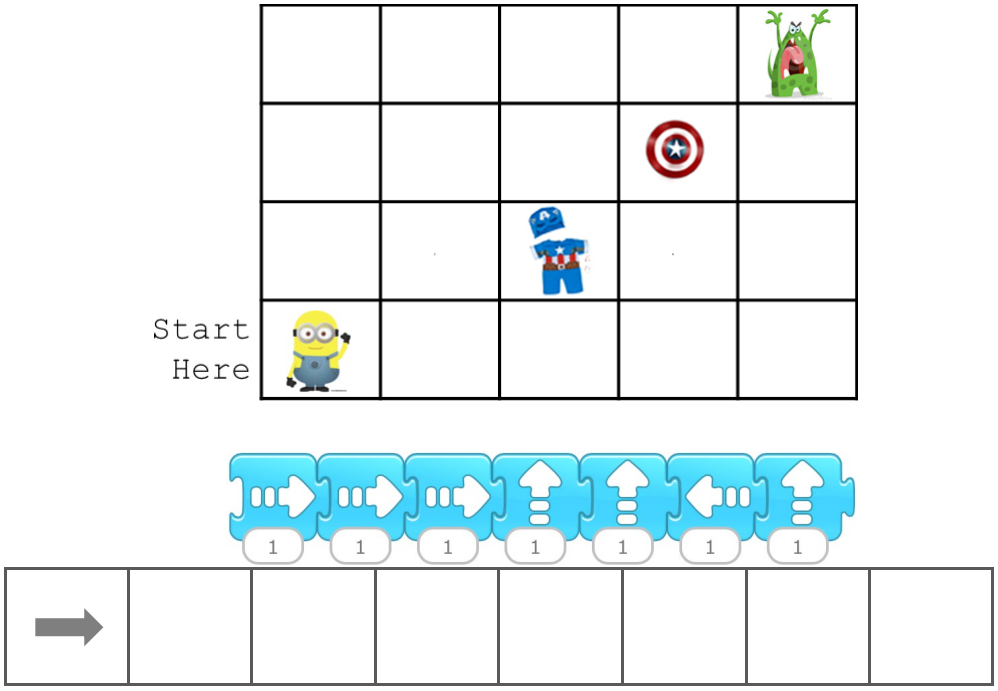
2. A turtle has to catch **Two Jewels**, then go to finish line.
Fill in FOUR BLANKS below.



3. Let's make a minion fully armed. 

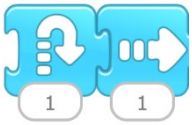
- Pick up a costume and a shield.
- Then meet a monster.

Find errors in the code and **Write a NEW CODE**.



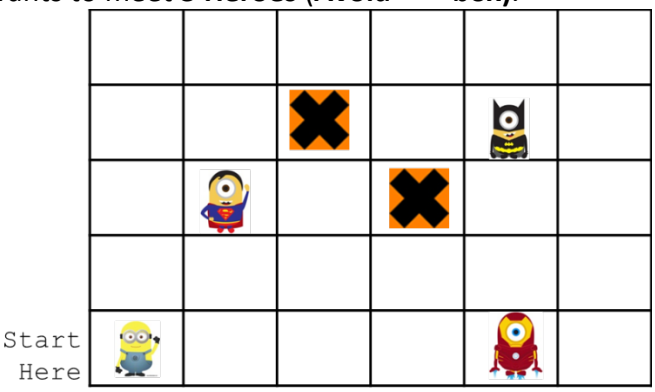
4. To get the hat, Jump pumpkins.

How Many Times the following code is repeated?

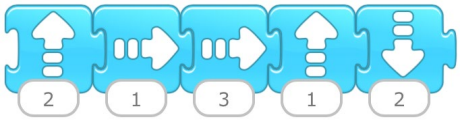


ANSWER: (____) times

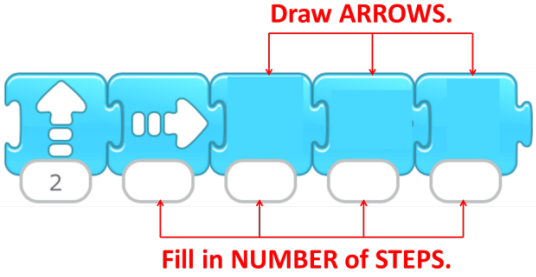
5. A minion really wants to meet **3 Heroes** (**Avoid**  **box)**.



(1) Is this code **RIGHT**? **Yes ( ) No ( )**



(2) **Revise Code** by drawing **ARROWS** or fill out **NUMBERS**.



Appendix E

Study Three: Self-Efficacy Survey

Please choose how well you can do each of the following <u>NOW</u>.

- I am able to make a coding project from an idea to a finished work.

No	I don't think so	Perhaps	I think so	Yes
1	2	3	4	5

- I can fix the code when something goes wrong.

No	I don't think so	Perhaps	I think so	Yes
1	2	3	4	5

- I feel confident that I can figure out how to use new features of an iPad coding app on my own.

No	I don't think so	Perhaps	I think so	Yes
1	2	3	4	5

- I have a very good understanding of how iPad coding app works.

No	I don't think so	Perhaps	I think so	Yes
1	2	3	4	5

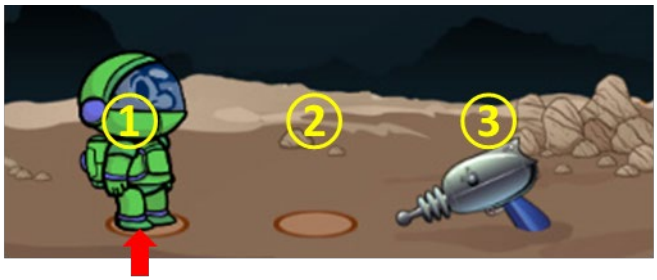
Appendix F

Study Three: Text-based Block Programming Test Items

moves “1 step”

1. If an astronaut follows code on the left, **WHERE would he be arrived**?
Choose one of the numbers.

Here’s Code

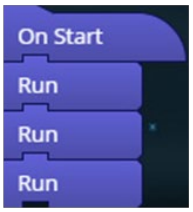


2. Which is the **RIGHT Code** for an astronaut gets the ray gun?
Choose one of the following.

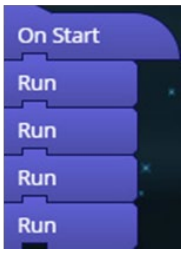
a.

b.

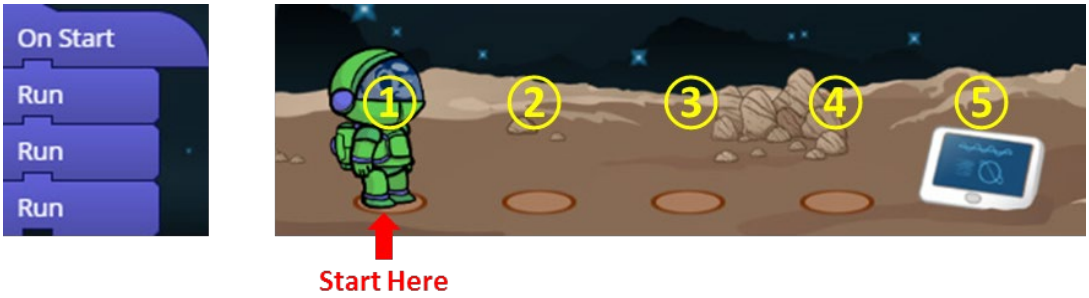
c.



d.



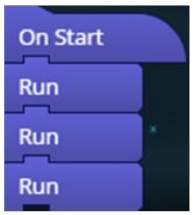
3. If follows the code, **WHERE would an astronaut be arrived**?
Choose one of the numbers.

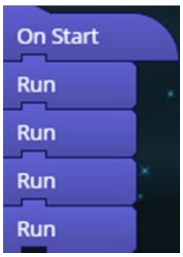


4. Which is the **RIGHT Code** to make an astronaut gets the ipad?
Choose one of the following.

a.


b.


c.


d.


5. **CHANGE Number** in the blank to get the power cell to power the spaceship.